EFFICIENT EMBEDDED COMPUTING

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

James David Balfour
May 2010

This dissertation is online at: http://purl.stanford.edu/nb912df4852

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

William Dally, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christoforos Kozyrakis

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Embedded computer systems are ubiquitous, and contemporary embedded applications exhibit demanding computation and efficiency requirements. Meeting these demands presently requires the use of application-specific integrated circuits and collections of complex system-on-chip components. The design and implementation of application-specific integrated circuits is both expensive and time consuming. Most of the effort and expense arises from the non-recurring engineering activities required to manually lower high-level descriptions of systems to equivalent low-level descriptions that are better suited to hardware realization. Programmable systems, particularly those that can be targeted effectively using high-level programming languages, offer reduced development costs and faster design times. They also offer the flexibility required to upgrade previously deployed systems as new standards and applications are developed. However, programmable systems are less efficient than fixed-function hardware. This significantly limits the class of applications for which programamble processors are an acceptable alternatives to application-specific fixed-function hardware, as efficiency demands often preclude the use of programmable hardware. With most contemporary computer systems limited by efficiency, improving the efficiency of programmable systems is a critical challenge and an active area of computer systems research.

This dissertation describes Elm, an efficient programmable system for high-performance embedded applications. Elm is significantly more efficient than conventional embedded processors on compute-intensive kernels. Elm allows software to exploit parallelism to achieve performance while managing locality to achieve efficiency. Elm implements a novel distributed and hierarchical system organization that allows software to exploit the abundant parallelism, reuse, and locality that are present in embedded applications. Elm provides a variety of mechanisms to assist software in mapping applications efficiently to massively parallel systems. To improve efficiency, Elm allows software to explicitly schedule and orchestrate the movement and placement of instructions and data.

This dissertation proposes and critically analyzes concepts that encompass the interaction of computer architecture, compiler technology, and VLSI circuits to increase performance and efficiency in modern embedded computer systems. A central theme of this dissertation is that the efficiency of programmable embedded systems can be improved significantly by exposing deep and distributed storage hierarchies to software. This allows software to exploit temporal and spatial reuse and locality at multiple levels in applications in order to reduce instruction and data movement.

Acknowledgments

First, I would like to thank my dissertation advisor, Professor William J. Dally, who has been great mentor and teacher. I benefited immensely from Bill's technical insight, vast experience, boundless enthusiasm, and steadfast support.

I would like to thank the other members of my dissertation committee, Professors Mark Horowitz, Christos Kozyrakis, and Boris Murmann. I had the distinct pleasure of having all three as lecturers at various times while a graduate student, and benefited greatly from their insights and experience. I imagine all will recognize some aspects of their teaching and influence within this dissertation.

I was fortunate to befriend some fantastic folks while at Stanford. In particular, I would like to thank Michael Linderman for years of engaging early morning discussions, and Catie Chang for years of engaging late night discussions and encouragement; my first few years at Stanford would not have been nearly as tolerable without their friendship. I would like the thank James Kierstead for several years of Sunday mornings spent discussing the plight of graduate students over waffles and coffee.

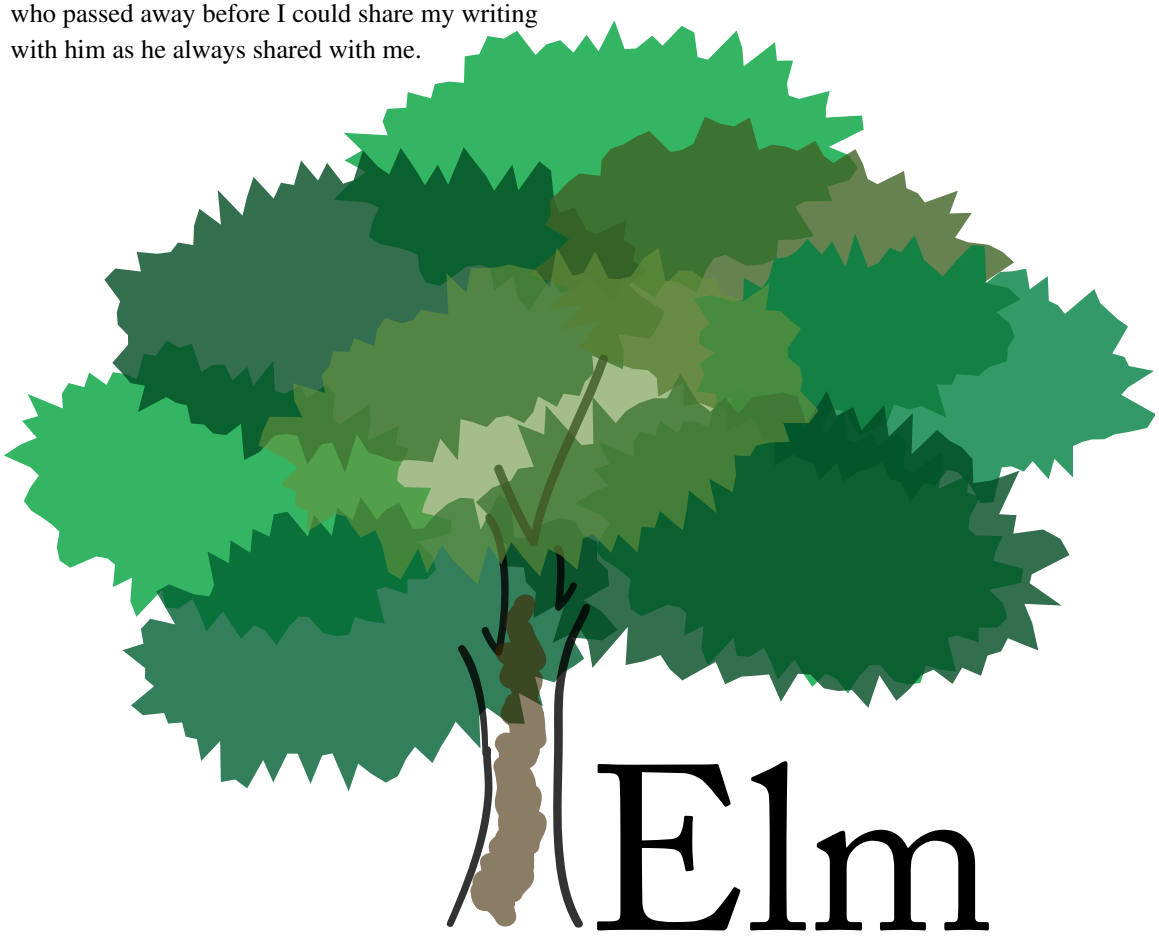
I particularly want to thank Tanya for her friendship, patience, support, assistance, and understanding. The last year would have been difficult without her.

I should like to acknowledge the other members of the concurrent VLSI architecture group at Stanford that I had the benefit of interacting with. In particular, I would like to acknowledge the efforts of the other members of the efficient embedded computing group. In approximate chronological order: David Black-Schaffer, Jongsoo Park, Vishal Parikh, R. Curtis Harting, and David Sheffield. I also had the pleasure of sharing an office with Ted Jiang, which I appreciated. Ted was always around and cheery on weekends, which made Sundays in the office more enjoyable, and kept the bookshelves well stocked.

While a student at Stanford, I was funded in part by a Stanford Graduate Fellowship that was endowed by Cadence Design Systems. I remain grateful for the support.

Finally, I would like to thank my family for all the love and support they have provided through the years.

Dedicated to the memory of Harold T. Fargey,
who passed away before I could share my writing
with him as he always shared with me.



Contents

List of Tables

List of Figures

Chapter 1

Introduction

Embedded computer systems are everywhere. Most of my contemporaries at Stanford carry mobile phone handsets that are more powerful than the computer I compiled my first program on. Embedded applications are adopting more sophisticated algorithms, evolving into more complex systems, and covering broader application areas as increasing performance and efficiency allow new and innovative technologies to be implemented in embedded systems.

Embedded applications will eventually exceed general-purpose computing in prevalence and importance. I argue that efficient programmable embedded systems are a critical enabling technology for future embedded applications. Improving the energy efficiency of computer systems is presently one of the most important problems in computer architecture. Perhaps the only other problem of similar importance is easing the construction of efficient parallel programs. As this dissertation demonstrates, data and instruction communication dominates energy consumption in modern computer systems. Consequently, improving efficiency necessarily entails reducing the energy consumed delivering instructions and data to processors and function units.

This dissertation describes Elm, an efficient programmable architecture for embedded applications. A central theme of this dissertation is that the efficiency of programmable embedded systems can be improved significantly by exposing deep and distributed storage hierarchies to software. This allows software to exploit temporal and spatial reuse and locality at multiple levels in applications in order to reduce instruction and data movement. Elm implements a novel distributed and hierarchical system organization that allows software to exploit the abundant parallelism, reuse, and locality that are present in embedded applications. Elm provides a variety of mechanisms to assist software in mapping applications efficiently to massively parallel systems. To improve efficiency, Elm allows software to explicitly schedule and orchestrate the movement of instructions and data, and affords software significant control over the placement of instructions and data.

Although this dissertation mostly contemplates efficiency, the relative complexity of mapping applications to Elm informed many of the architectural decisions. Mapping applications from high level descriptions of systems to detailed implementations imposes substantial engineering and time costs. The Elm architecture attempts to simplify the complex and arduous task of compiling and mapping software that must satisfy real-time performance constraints to programmable architectures. Elm is designed to allow applications to be

compiled from high-level languages using optimizing compilers and efficient runtime systems, eliminating the need for extensive development in low-level languages. The Elm architectures is specifically designed to allow software to embed strong assertions about dynamic system behavior and to construct deterministic execution and communication schedules.

1.1 Embedded Computing

It has become rather difficult to differentiate clearly between embedded computer systems and general-purpose computer systems. Embedded systems continue to increase in complexity, scope, and sophistication. Advances in semiconductor and information technology made massive computing capability both inexpensive and pervasive. The following paragraphs describe ways in which embedded applications and embedded computer systems differ significantly from general-purpose computer systems.

Power and Energy Efficiency — Embedded systems have demanding power and energy efficiency constraints. Active cooling systems and packages with superior thermal characteristics increase system costs. Many embedded systems operate in harsh environments and cannot be actively cooled. The limited space within enclosures may preclude the use of heat sinks. In mobile devices such as cellular telephone handsets, energy efficiency is a paramount design constraint because devices rely on batteries for energy and must be passively cooled.

Performance and Predictability — Contemporary embedded applications have demanding computation requirements. For example, signal processing in a 3G mobile phone handsets requires upwards of 30 GOPS for a 14.4 Mbps channel, while signal processing requirements for a 100 Mbps OFDM [110] channel can exceed 200 GOPS [131]. Future applications will present even more demanding requirements as more sophisticated communications standards, compression techniques, codecs, and algorithms are developed. In addition, many applications impose real-time performance requirements on embedded computer systems. For example, a digital video decoder must deliver frames to the display at a rate that is consistent with the video stream, as there is little buffering in such systems to decouple decoding from presentation. Control systems, such as those found in automotive and telecommunications applications, impose even more demanding real-time constraints.

Designing systems to satisfy real-time performance constraints is challenging. Designers must reason about the dynamic behavior of complex systems with many interacting hardware and software components. The design process is simplified significantly when hardware and software is constructed to allow reasonably strong assertions to be made about the performance of components perform and how they interact. Predicting the performance of modern processors is notoriously difficult because structures such as caches and branch prediction units that improve performance introduce significant amounts of variability into the execution of software. Embedded processors often provide mechanisms that allow software to eliminate some of the variability, such as caches that allow specific cache blocks and ways to be locked. However, such mechanisms are typically exposed to software in such a way that programmers must program at a very low-level of abstraction. Application-specific fixed-function hardware can often be designed to deliver deterministic behavior

that is readily analyzed and understood, though this comes at the expense of designers explicitly specifying the behavior of the hardware in precise detail.

For most embedded applications with real-time constraints, delivering performance and capabilities beyond those required to satisfy the real-time performance requirements offers little benefit. Rather than optimizing for computational performance, the design objective is often to minimize costs and reduce energy consumption.

Cost and Scalability — Embedded applications present a broad range of performance needs and cost allowances. Components within a cellular network such as base stations may be expensive because the systems are expected to be deployed for many years. Network operators prefer for these systems to be extensible and upgradeable so that deployed equipment can be updated as new technologies and standards are developed, and operators are accordingly willing to pay a premium for upgradeable systems that preserve their capital investments. Components in edge and client devices such as cellular handsets must be inexpensive as they are typically discarded after a few years.

Cost and energy efficiency considerations favor integrating more components of a system on a single die. Historically, costs associated with fabricating, testing, and packaging chips have been dominant in most embedded systems. The modest non-recurring engineering and tooling costs associated with the design, implementation, verification, and production of mask sets were amortized over the large production volumes needed to meet the demand for the consumer products that account for most embedded systems. Low volume systems, such as those used in defense and medical applications, exhibited different cost structures, and often had considerably greater selling prices.

However, it has become increasingly more difficult and expensive to design, implement, and verify chips in advanced semiconductor processes. Because programmable systems are presently not efficient enough for demanding embedded applications to be implemented in software, many embedded systems require significant amounts of application-specific fixed-function logic to perform computationally demanding tasks. The design of these systems involves the manual partitioning and mapping of applications to many different hardware and software components, and intensive hardware design and validation efforts. The implementation complexity and effort associated with developing application-specific hardware impose significant engineering costs, and the time and effort required to implement and verify application-specific fixed-function logic increases with the scale and complexity of a system. Furthermore, the inflexibility of deployed systems that rely on application-specific fixed-function hardware increases the cost of implementing new standards and slows their adoption because existing infrastructure must be replaced.

The significant cost of designing new systems often discourages the development, adoption, and deployment of innovative applications, technologies, algorithms, protocols, and standards. For example, the expenses associated with developing novel medical equipment can limit the deployment and accessibility of important medical devices, as the potential markets for new devices may not be large enough to adequately amortize development costs in addition to those costs associated with regulatory approval and clinical trials. Because designs require very large volumes to justify such engineering costs, designs will be limited to individual products with large volumes and families of related products with large aggregate volumes.

This favors configurable and programmable systems, such as programmable processors, field programmable gate arrays, and systems that integrate fixed-function hardware and programmable processors on a single die. Unfortunately, partitioning applications between software and hardware requires programming at low-levels of abstraction to expose the hardware capabilities to software, which increases application software development costs and development times: the design and verification of a complex system-on-chip can require hundreds of engineer-years [102] and incur non-recurring engineering costs in excess of $20 M – $40 M.

1.2 Technology and System Trends

Billions of dollars in research and development have driven decades of steady improvements in semiconductor technology. These improvements have allowed manufacturers to increase the number of transistors that can be reliably and economically integrated on a single chip. Transistor densities have historically doubled approximately every two years with the introduction of a new CMOS process generation, and densities at present allow billions of transistors to be fabricated reliably on a single die.

Until recently, most computer architecture research focused on techniques for using additional transistors to improve the single-thread performance of existing applications. Processors were designed with deeper pipelines to allow clock frequencies to increase faster than transistor switching speeds improved with scaling, while increasingly more complex and aggressive microarchitectures were used to improve instruction execution rates. Many of the additional transistors were used to implement impressive collections of execution units that allowed increasing numbers of instructions to be in flight and complete out of order. As pipeline depths and relative memory latencies increased, more of the additional transistors were used to implement the larger caches, more complex superscalar issue logic, and more elaborate predictors that were needed to deliver an adequate supply of instructions and data to the ravenous execution units. Fundamental physical constraints and basic economic constrains, such as the cost of building cooling systems to dissipate heat from integrated circuits, limit the ability of these techniques to deliver further improvements in sequential processor performance without continued voltage scaling.

Interconnect and Communication

Interconnect performance benefits less than transistor performance from scaling. Consequently, improvements in wire performance fail to maintain pace with improvements in transistor performance, and communication becomes relatively more expensive in advanced technologies. Classic linear scaling causes interconnect resistance per unit length to increase while capacitance per unit length remains approximately constant [36]. Because local interconnect lengths decrease with lithographic scaling, classical scaling causes local interconnect capacitances to decrease and resistance to increase such that the characteristic *RC* response time of scaled local wires remains approximately constant. The reduction in capacitance improves energy efficiency and the increase in wire densities delivers greater aggregate bandwidth, but the increase in resistance results in wires that appear slow relative to a scaled transistor. To compensate, the thickness of metal layers has often been scaled at a slower rate to reduce wire resistance [68], and wires in contemporary CMOS

processes are often taller than they are wide. However, effects such as carrier surface scattering cause conductance to degrade more rapidly as scaled wires become narrower [67]. Wires that span fixed numbers of transistor gate pitches shrink with lithographic scaling, and the resulting reduction in capacitance compensates for some of the increase in resistance. Though repeater insertion and the use of engineered wires can partially ameliorate the adverse consequences of interconnect scaling[67], scaling effectively renders communication relatively more expensive than computation in advanced technologies. Somewhat worryingly, interconnect conductance decreases significantly when the mean free path becomes comparable to the thickness of a wire.

Scaling, Power, and Energy Efficiency

Historically, semiconductor scaling simultaneously improved the performance, density, and energy efficiency of integrated circuits by scaling voltages with lithographic dimensions, commonly referred to as Dennard scaling [36]. Theoretically, Dennard scaling maintains a constant power density while increasing switching speeds deliver faster circuits and increasing transistor densities allow more complex designs to be manufactured economically. Unfortunately, the subthreshold characteristics of devices tend to degrade when threshold voltages are reduced as devices are scaled. The reduction in threshold voltages allows supply voltages to be lowered without degrading device performance, thereby reducing switching power dissipation and energy consumption. However, the reductions in transistor threshold voltages specified for Dennard scaling result in increased subthreshold leakage currents, which effectively limits the extent to which threshold voltages can be usefully reduced. Leakage currents presently contribute a significant fraction of the total energy consumption and power dissipation in modern processes designed for high-performance integrated circuits.

With the performance and efficiency of many contemporary systems limited by power dissipation, poor subthreshold characteristics seem likely to prevent further significant threshold voltage scaling [47, 69]. Increasing device variation and mismatch, which results from scaling [115], exacerbates the problem, as it becomes increasingly difficult to control device characteristics such as threshold voltages [117]. Without continued transistor threshold voltage scaling, further supply voltage reductions will reduce device switching speeds, effectively trading reductions in area efficiency for improvements in energy efficiency. High-performance systems are limited by their ability to deliver and dissipate power; many of the future improvements in the performance of these systems will depend on improvements in efficiency, so that computation demands less power. Similarly, mobile and embedded systems are limited by cost, power, and battery capacity. Consequently, energy efficiency has become a paramount design consideration and constraint across all application domains.

1.3 A Simple Problem of Productivity and Efficiency

Simple in-order processor cores, such as those used in low-power embedded applications, are more efficient than the complex cores that dominate modern high-performance computer architectures. With less elaborate hardware used to orchestrate the delivery of instructions and data, more of the energy consumed in

simple cores is used to perform useful computation. However, merely integrating large numbers of simple conventional cores will not deliver efficiencies that are anywhere near sufficient for demanding embedded applications that currently require application-specific integrated circuits. As data presented in this dissertation demonstrates, most of the energy consumed in microprocessors is used to move instructions and data, not to compute. Consequently, programmable systems deliver poor efficiency relative to dedicated fixed-function hardware. The efficiency of programmable embedded systems is particularly poor compared to fixed-function implementations because the operations used in embedded applications, fixed-point arithmetic and logic operations, are inexpensive compared to delivering instructions and data to the function units: performing a 32-bit fixed-point addition requires less energy than reading its operands from a conventional register file and writing back its result, as Table 1.1 shows. Because interconnect benefits less than logic from improvements in semiconductor technology, the interconnect-dominated memories and buses that deliver instructions and data to the function units consume an increasing fraction of the energy. As discussed previously, this imbalance will not improve with advances in semiconductor technology. Instead, architectures must be designed specifically to reduce communication, capturing more instruction and data bandwidth in simple structures that are close to function units, to deliver improvements in efficiency.

This dissertation focuses mostly on technologies for improving efficiency. The efficient embedded computing project in which Elm evolved developed technologies that addressed both the problem of programming massively parallel systems and improving system efficiency. Both are challenging and ambitious research problems. The architecture I describe attempts to allow software to construct deterministic execution schedules to simplify the arduous task of compiling parallel software to meet real-time performance constraints.

1.4 Collaboration and Previous Publications

This dissertation describes work that was performed as part of an effort by members of the Concurrent VLSI Architecture research group within the Stanford Computer Systems Laboratory to develop architecture, compiler, and circuit technologies for efficient embedded computer systems [12, 13, 19, 33]. Jongsoo Park was responsible for the implementation and maintenance of the compilers we developed as part of the project. The benchmarks that appear in this dissertation were written and maintained by Jongsoo Park, David Black-Shaffer, Vishal Parikh, and myself.

1.5 Dissertation Contributions

This dissertation makes the following contributions.

Efficiency Analysis of Embedded Processors — This dissertation presented an analysis of the energy efficiency of a contemporary embedded RISC processor, and argued that inefficiencies inherent in conventional RISC architectures will prevent them from delivering the efficiencies demanded by contemporary and emerging embedded applications.

Datapath Operations		**Relative Energy**
32-bit addition	520 fJ	1×
16-bit multiply	2,200 fJ	4.2×
32-bit pipeline register	330 fJ	0.63×
Embedded RISC Processor		**Relative Energy**
Register File [32 entries 2R+1W]		
32-bit read	250 fJ	0.48×
32-bit write	470 fJ	0.90×
Data Cache [2 KB 4-way set associative]		
32-bit load	3,540 fJ	6.8×
32-bit store	3,530 fJ	6.8×
miss	1,410 fJ	2.7×
Instruction Cache [2 KB 4-way set associative]		
32-bit fetch	3,500 fJ	6.8×
miss	1,410 fJ	2.7×
128-bit refill	9,710 fJ	19×
Instruction Filter Cache [64-entry direct-mapped]		
32-bit fetch	990 fJ	1.9×
miss	430 fJ	0.82×
128-bit refill	2,560 fJ	4.9×
Instruction Filter Cache [64-entry fully-associative]		
32-bit fetch	1,320 fJ	2.5×
miss	980 fJ	1.9×
128-bit refill	2,610 fJ	5.0×
Execute add Instruction [Lower Bound]	**5,320 fJ**	10.2×

Table 1.1 – Estimates of Energy Expended Performing Common Operations. The data were derived from circuits implemented in a 45 nm CMOS process that is tailored for low standby-power applications. The process uses thick gate oxides to reduce leakage, and a nominal supply of 1.1 V to provide adequate drive current. The energy used to execute an add instruction includes fetching an instruction, reading two operands from the register file, performing the addition, writing a pipeline register, and writing the result to the register file.

Instruction Registers — This dissertation introduced the concept of instruction registers. Instruction registers are implemented as efficient small register files that are located close to function units to improve the efficiency at which instructions are delivered.

Operand Registers — This dissertation described an efficient register organization that exposes distributed collections of operand registers to capture instruction-level producer-consumer locality at the inputs of function units. This organization exposes the distribution of registers to software, and allows software to orchestrate the movement of operands among function units.

Explicit Operand Forwarding — This dissertation described the use of explicit operand forwarding to improve the efficiency at which ephemeral operands are delivered to function units. Explicit operand forwarding uses the registers at the outputs of function units to deliver ephemeral data, which avoids the costs associated with mapping ephemeral data to entries in the register files.

Indexed Registers — This dissertation presented a register organization that exposes mechanisms for accessing registers indirectly through indexed registers. The organization allows software to capture reuse and locality in registers when data access patterns are well structured, which improves the efficiency at which data are delivered to function units.

Address Stream Registers — This dissertation presented a register organization that exposes hardware for computing structured address streams as address stream registers. Address stream registers improve efficiency by eliminating address computations from the instruction stream, and improve performance by allowing address computations to proceed in parallel with the execution of independent operations.

The Elm Ensemble Organization — This dissertation presented an Ensemble organization that allows software to use collections of coupled processors to exploit data-level and task-level parallelism efficiently. The Ensemble organization supports the concept of processor corps, collections of processors that execute a common instruction stream. This allows processors to pool their instruction registers, and thereby increase the aggregate register capacity that is available to the corps. The instruction register organization improves the effectiveness of the processor corps concept by allowing software to control the placement of shared and private instructions throughout the Ensemble.

This dissertation presented a detailed evaluation of the efficiency of distributing a common instruction stream to multiple processors. The central insight of the evaluation is that the energy expended transporting operations to multiple function units can exceed the energy expended accessing efficient local instruction stores. In exchange, the additional instruction capacity may allowing additional instruction working sets to be captured in the first level of the instruction delivery hierarchy. Consequently, the improvements in efficiency depend on the characteristics of the instruction working set. Elm allows software to exploit structured data-level parallelism when profitable by reducing the expense of forming and dissolving processor corps.

The Elm Memory System — This dissertation presented a memory system that distributes memory throughout the system to capture locality and exploit extensive parallelism within the memory system. The memory system provides distributed memories that may be configured to operate as hardware-managed caches or software-managed memory. This dissertation also described how simple mechanisms provided by the hardware allow the cache organization to be configured by software, which allows a virtual cache hierarchy to be adjusted to capture the sharing and reuse exhibited by different applications. The memory system exposes mechanisms that allow software to orchestrate the movement of data and instructions throughout the memory system. This dissertation described mechanisms that are integrated with the distributed memories to improve the handling of structured data movement within the memory system.

1.6 Dissertation Organization

Before proceeding it seems prudent to sketch an outline of the remainder of this dissertation. Chapter 2 presents an analysis of contemporary embedded architectures and argues that conventional architectures will not provide the efficiencies demanded by contemporary and emerging embedded applications. Chapter 3

introduces the Elm architecture, and presents the central unifying themes and ideas of the dissertation. Chapters 4 through 6 describe aspects of the Elm processor architecture that improve the efficiency at which instructions and operands are delivered to function units. Chapter 7 describes how collections of processors are assembled into Ensembles, and discusses various issues related to using collections of processors to exploit data-level and task-level parallelism. Chapter 8 describes the Elm memory system. Chapter 9 concludes the dissertation, summarizing contributions and suggesting future research directions.

Three appendices provide additional materials that would interrupt the flow of the thesis were they presented within the dissertation proper. Appendix A describes the experimental methodology and simulators that were used to obtain the data presented throughout this dissertation. Appendix B describes the kernel and benchmark codes that are discussed throughout the dissertation and are used to evaluate various ideas and mechanisms. Appendix C provides a listing of instructions that are somewhat unique to Elm and a detailed description of the Elm instruction set architecture. The reader may find these useful when contemplating various assembly code listings that appear in this dissertation.

Chapter 2

Contemporary Embedded Architectures

Contemporary embedded computer systems comprise diverse collections of electronic components. Advances in semiconductor scaling allow complex systems to be integrated one a single system-on-chip. Most system-on-chip components comprise heterogeneous collections of microprocessors, digital signal processors, application-specific fixed-function hardware accelerators, memory controllers, and input-output interfaces that allow the chip to communicate with other devices. In most contemporary embedded systems, the majority of the computation capability is provided by a few system-on-chip components.

It is reasonable to wonder whether we could replace the fixed-function hardware with collections of simple processors similar to the RISC processors that are prevalent in contemporary embedded systems. The processors would perhaps be organized as a multicore or manycore processor system-on-chip. It is possible for such an architecture to deliver sufficient arithmetic bandwidth to satisfy the demands of most applications, as a tiled collection of small processors can provide a large aggregate number of function units. However, I argue in this chapter that the energy efficiency of simple RISC processors is insufficient for such a system to deliver acceptable power and energy efficiency.

This chapter provides a brief examination of existing embedded computer systems. We begin with a case study of a simple and efficient embedded RISC processor that is representative of contemporary commercial processors; we consider its efficiency, and examine where and how energy is expended. We then discuss strategies and technologies that have been proposed for improving energy efficiency. We conclude with a survey of a few relevant architectures that have used these technologies to improve efficiency.

2.1 The Efficiency Impediments of Programmable Systems

Analyzing how area and energy is consumed in simple RISC processors provides significant insight into the efficiency limitations of conventional programmable processors. Table 2.1 lists significant attributes of a very simple embedded processor. The processor is derived from a synthesizable implementation of 32-bit SPARC processor designed for embedded applications [51]. We modified the design to remove everything except the integer pipeline, instruction and data caches, and those parts of the memory controller that are needed to

Instruction Set Architecture	SPARC V8 [32-bit]
Clock Frequency	500 MHz
Pipeline	7 Stage Integer Pipeline
Hardware Multiplier	16-bit × 16-bit with 40-bit Multiply-Accumulate Unit
Register Windows	2 windows
Register File	40 registers [2R + 1W]
Instruction Cache	4 KB Direct-Mapped
Data Cache	4 KB Direct-Mapped

Table 2.1 – Embedded RISC Processor Information. The processor is based on a synthesizable 32-bit SPARC core. The processor contains an integer pipeline, data and instruction cache controllers, and a basic memory controller interface. Many of the applications we consider in this dissertation operate on 16-bit operations. The integer unit provides a 16-bit hardware multiplier, which is adequate for most operations, and implements 32-bit multiplication instructions as multi-cycle operations.

fetch instructions and data on cache misses. With the hardware needed to support sophisticated architectural features such as virtual memory removed, what remains is a design that approximates an efficient minimal embedded RISC processor. We further modified the implementation to support local clock gating throughout the processor.

Area Efficiency

Figure 2.1 shows the area occupied by the processor. The entire design occupies a region that is about 0.5 mm by 0.3 mm. The area reported is measured after the design is placed and routed. As the data presented in the figure clearly illustrate, despite the relative simplicity of the architecture and implementation, the processor area is dominated by circuits that are used to stage instructions and data. The integer core contributes 22.5% of the area, the register file 11.0%, and the instruction and data caches contribute the remaining 66.6%. About 53% of the standard cell area in the integer core is contributed by flip-flops and latches, most of which are used to coordinate the movement of instructions to the function units and to stage operands and results. The tag and data arrays are implemented as high-density SRAM modules that are compiled and provided by the foundry. Despite the density and small capacities of the arrays, the area of the caches is dominated by the tag and data arrays, which consume 56.3% of the entire area. The cache and memory controllers occupy about 15.4% of the cache area.

To evaluate the area efficiency, we need to account for the rate at which the processor completes application work. The number of instructions executed per cycle (IPC) provides a reasonable estimate of the effective processor utilization, though it does not account for how efficiently application operations may be mapped to the processor operations exposed by the instruction set. Figure 2.2 shows the average number of instructions executed per cycle for several common embedded kernels. The kernels are described in detail in Appendix A. The reported data were measured after the persistent instruction and data working sets are captured in the caches; any loss of performance due to cache misses is introduced by compulsory data misses. The processor has a peak instruction execution rate of one instruction per cycle, and the kernels

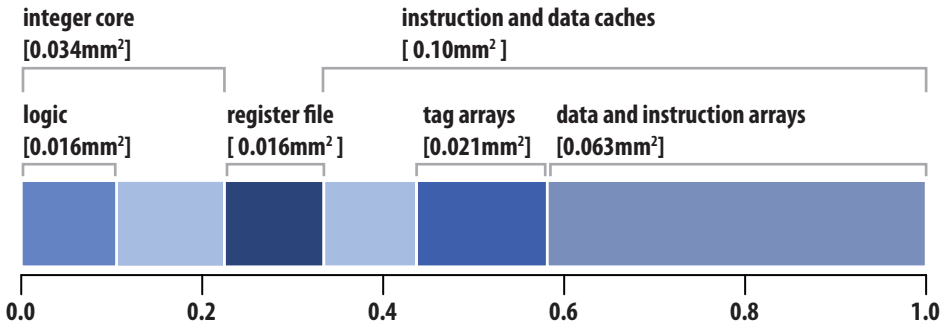


Figure 2.1 – **Embedded RISC Processor Area.** The area reported is measured after the design is placed and routed. The design occupies a region that is about 0.5 mm by 0.3 mm.

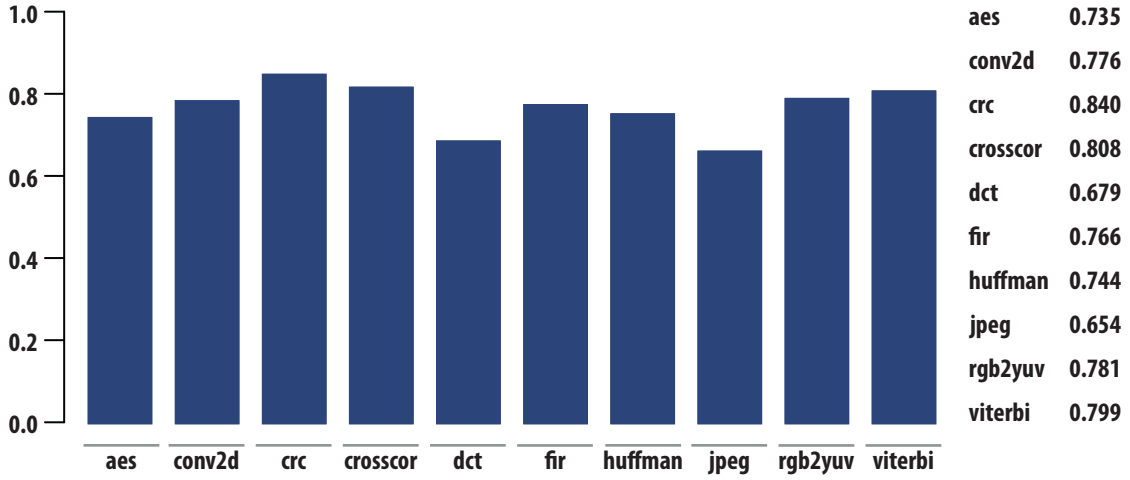


Figure 2.2 – **Average Instructions Per Cycle for an Embedded RISC Processor.** The processor has a peak instructions per cycle (IPC) of 1 instruction per cycle. The reported average number of instructions executed per cycle is measured after the system reaches a steady state of operation, and reflects the behavior that would be observed when a single kernel is mapped to a processor and executes for a long period of time.

are carefully coded and optimized to reduce the number of avoidable pipeline hazards and stalls. However, compulsory data cache misses, branch delay penalties, and true data dependencies result in the processor sustaining between 65% and 84% of peak instruction throughput.

Energy Efficiency

In order to assess the energy efficiency of the processor, we need to understand where energy is expended and the contribution of different operations to the energy consumption. We first consider the relative contribution of the processor core and caches to the aggregate energy consumption. Figure 2.3 shows the distribution of energy consumed in the caches and processor core. The data have been normalized within each kernel to

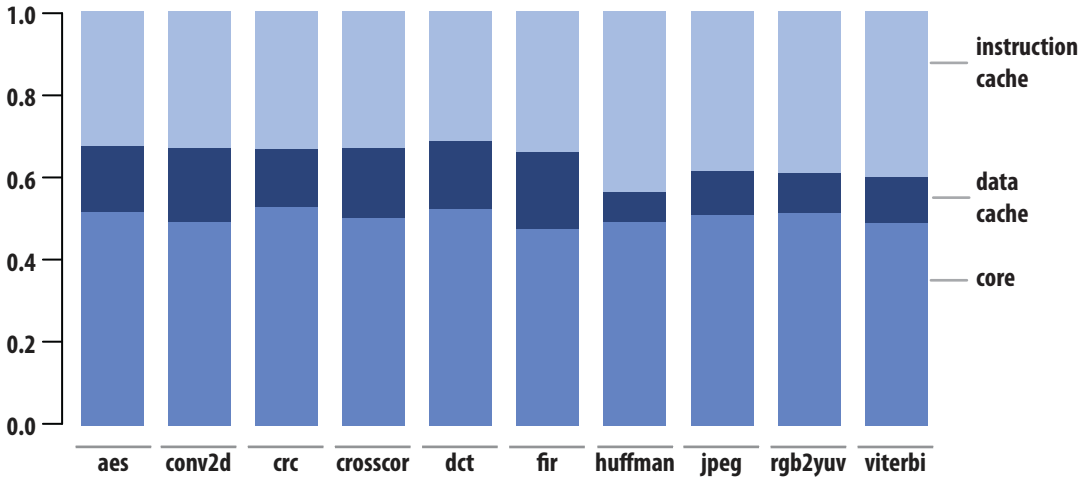


Figure 2.3 – **Energy Consumption by Module for an Embedded RISC Processor.** The cache components include the cache data and tag memory arrays and the cache controllers. The core component includes the integer pipeline and all other control logic. Both components include energy consumed in the clock tree, with energy consumed within shared levels apportioned relative to the clock load within each module.

illustrate clearly the contribution of each component. As the figure illustrates, the composition of the energy consumption is somewhat insensitive to the computation performed by a kernel. The cache energies include both the energy consumed in the storage arrays and the cache controllers. Though the instruction and data caches are similar in structure and identical in capacity, the data cache is more expensive to access because the data cache controller implements additional hardware to support stores and must track which cache blocks have been modified. However, the instruction cache consumes more energy because it is accessed more often. The amount of energy consumed in the processor core is similar to the energy consumed in the caches. This results in part from the small 4 KB caches used in the evaluation.

The composition of the dynamic instruction streams provides some insight into the relative contribution of different operations to the energy expended by the processor. It also explains the differences in the relative contributions of the caches. Figure 2.4 shows the frequency of different instruction types in the dynamic instruction streams comprising the kernel executions. The control component corresponds to instructions that affect control flow, such as jumps and conditional branches. The memory component corresponds to instructions that transfer data between registers and memory, such as load and store instructions. The arithmetic and logic component corresponds to compute instructions that operate on data stored in registers.

As Figure 2.4 illustrates, the kernels are dominated by compute and memory instructions. Arithmetic and logic instructions alone account for approximately 70% of the instructions that are executed across the kernels. The small contribution of control instructions results from the compiler aggressively unrolling loops with predictable loop bounds. The notable exceptions are the **huffman** and **jpeg** kernels, both of which contain significant loops with data-dependent iteration counts and data-dependent conditional branches.

The distribution of instruction types provides an incomplete measure for assessing the energy efficiency

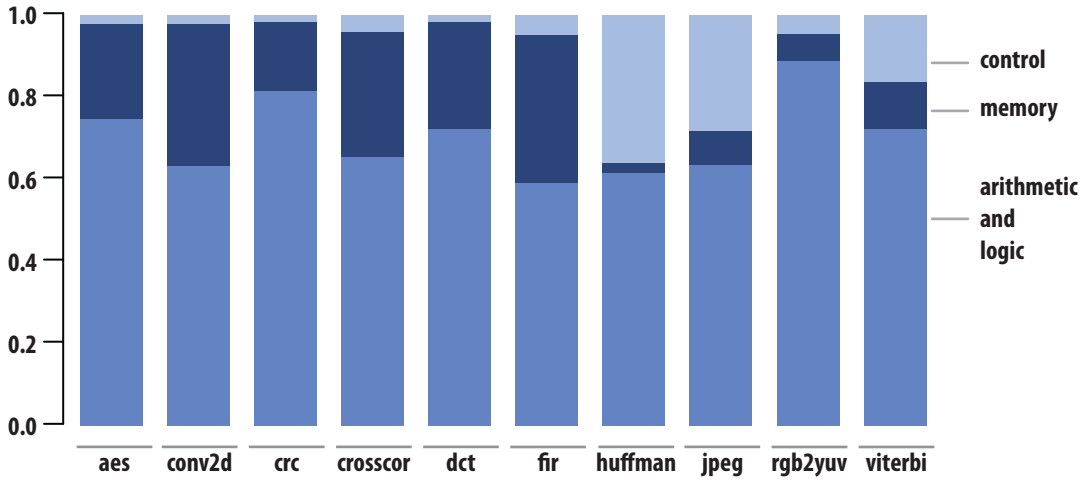


Figure 2.4 – **Distribution of Instructions by Operation Type in Embedded Kernels.** The control component includes control flow instructions, the preponderance of which are conditional branches and jump instructions. The memory component includes data transfer instructions that access memory, which are almost exclusively load and store instructions. The arithmetic and logic component includes all arithmetic and logic instructions, which are dominated by integer arithmetic and logic operations. The processor implements a multiply-and-accumulate instruction, which encodes two basic arithmetic operations. The multiply-and-accumulate instruction is used extensively in the arithmetically intensive kernels.

of the processor because it does not account for how different instructions contribute to the computation demanded by the kernels. Many of the arithmetic and logic instructions encode operations that are not fundamental to the kernels, but a consequence of their implementation. For example, arithmetic and logic instructions may appear in instruction sequences that implement control and memory access operations that cannot be encoded directly in the instruction set. Data references and control statements in programming languages often map to sequences of instructions. For example, a compiler may map an effective address calculation to an instruction sequence comprising an arithmetic instruction followed by a load instruction. It is important that we differentiate operations that are fundamental to a computation from computations that are a consequence of the architecture a computation is mapped to.

To asses correctly the impact of the processor architecture on its energy efficiency, we implemented a compiler data flow analysis to classify instructions as control and memory operations based on how the values the instructions compute are used rather than the operations instructions perform. The data flow analysis can analyze assembly code listings produced by other compilers. It uses the results of a reaching definitions analysis to propagate data flow information backwards from control flow and memory instructions. The reaching definitions analysis follows variables that are stack allocated and temporarily spilled and restored during a procedure invocation through spills and restores to fixed memory locations. The analysis identifies instructions that compute condition operands to control flow operations and address operands to memory operations. The analysis propagates data flow information backwards through arithmetic and logic instructions to identify sets of dependent arithmetic and logic instructions that compute control and address

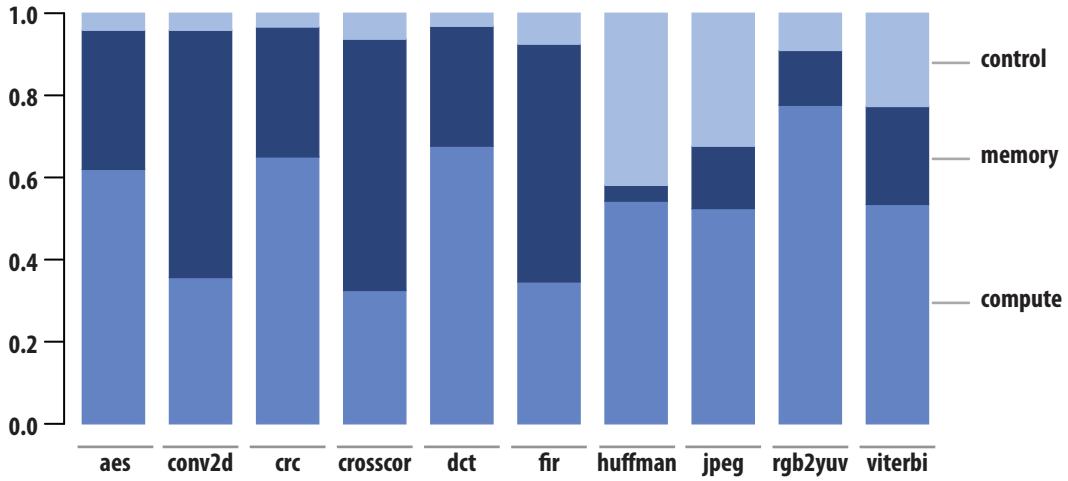


Figure 2.5 – **Distribution of Instructions within Embedded Kernels.** The control component includes control flow instructions and those arithmetic and logic instructions that compute operands to control flow instructions. The memory component includes data transfer instructions that access memory and those arithmetic and logic instructions that provide address operands to data transfer instructions. The compute component includes arithmetic and logic instructions that implement computations demanded by the kernels. An arithmetic or logic instruction that defines a value that is both fundamental to the computation of a kernel and used by a control or memory operation is designated a compute instruction. An arithmetic or logic instruction that defines a value that is an operand to both a control and memory operation is designated a control instruction.

operands. Arithmetic and logic instructions that define values that are used exclusively as condition and address operands or exclusively to compute condition and address operands are classified as control and memory operations.

The results of the analysis appear in Figure 2.5. The data illustrated in the figure provide a more detailed accounting of the composition of the instructions comprising the kernels. Compute operations are encoded in 53% of the instructions on average. The **crosscor** kernel is dominated by memory operations, and only 32% of its instructions encode compute operations. The **rgb2yuv** kernel has very simple control and small data working sets that the compiler maps to the register file, and 77% of its instructions encode compute operations.

To assess the energy efficiency of the processor, we also need to account for the impact of microarchitecture on the energy consumed in the processor core. A significant component of the energy expended in the processors is used to deliver operands to function units and control signals decoded from instructions to data-path control points. Though necessary to execute instructions, such energy is an overhead imposed by the programmability afforded by the architecture.

Figure 2.6 shows a detailed analysis of the contribution of instruction delivery, data delivery, and compute to the aggregate energy consumption of each kernel. The instruction delivery energy reflects the preponderance of the energy used to deliver instructions to the data-path control points. Significant components include the energy used to access the instruction cache and transfer instructions to the processor, energy expended

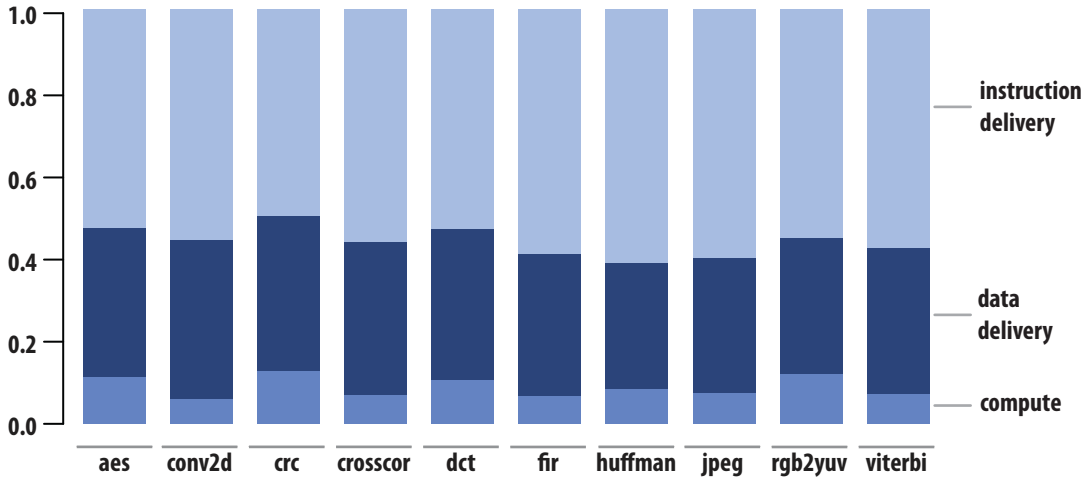


Figure 2.6 – **Distribution of Energy Consumption within Embedded Kernels.** Instruction delivery and data delivery contribute the majority of the aggregate energy consumption for each kernel.

in the instruction fetch and decode logic, and in flip-flops that implement interstage pipeline registers that capture instruction and control signals. The data delivery energy reflects the preponderance of the energy used to transfer data the function units. Significant components include the energy expended accessing the data cache and transferring data between the processor core and data cache, energy expended accessing the register file, driving results back to the register file, and in flip-flops that implement interstage pipeline registers that capture operands and results. The compute component measures the energy used in the function units to complete operations demanded by a kernel. Perhaps the most salient and significant aspect of the data is the small component of the energy contributed by compute, which varies from 6.0% to 12.8%.

Table 2.2 provides the average energy per instruction for each of the kernels. The table lists separately the component of the average energy per instruction that we attribute to instruction delivery, data delivery, and compute for each of the kernels. Differences between the kernels reflects differences in the computations and working sets.

2.2 Common Strategies for Improving Efficiency

We can reason about many strategies for improving efficiency as mechanisms that attempt to reduce instruction and data bandwidth demand somewhere in the system. Some attempt to shift more of the bandwidth demand closer to the function units, while others attempt to eliminate instruction and operand references. Often, these schemes are described as exploiting some form of reuse and locality to keep more control and data bandwidth close to the function units. The following are some of the most pervasive and effective techniques.

	aes	conv2d	crc	crosscor	dct	fir	huffman	jpeg	rgb2yuv	viterbi
Instruction	34.9	33.2	30.3	32.2	37.7	34.0	35.4	39.9	32.8	32.5
Data	23.9	23.0	22.8	21.2	26.1	19.8	17.7	21.85	19.5	19.9
Compute	7.4	3.6	7.9	4.1	7.5	3.9	4.9	5.0	7.2	4.1
	66.2	**59.8**	**61.0**	**57.5**	**71.3**	**57.7**	**58.0**	**66.7**	**59.4**	**56.5**
Other	5.6	7.5	5.3	6.7	13.5	6.2	8.1	21.0	11.7	12.3
Total	**71.8**	**67.3**	**66.3**	**64.2**	**84.8**	**63.9**	**66.1**	**87.0**	**71.1**	**68.7**

Table 2.2 – Average Energy Per Instruction for an Embedded RISC Processor. The energy is reported in picojoules and is computed by averaging over all of the instructions comprising a kernel. The average energy per instruction captures the energy consumed performing the computation that is fundamental to a kernel, energy consumed transferring instructions and data to function units, and energy consumed orchestrating the movement of instructions and data. The Instruction row lists the energy that is consumed delivering instructions to the function unit. The Data row lists the energy consumed delivering data to the function unit. The Compute row lists the energy consumed in the function units to execute those instructions that perform operations that are intrinsic to a kernel, as opposed to those operations that are executed to coordinate the movement and sequencing of instructions and data. The Other row lists energy contributed by when the processor is stalled. Mechanisms such as leakage current and activity in the clock network when the processor is stalled contribute a significant component of the average energy per instruction. Most of the flip-flops in the design are clock-gated, and consequently there is little activity near the leaf elements when the processor is stalled; however, nodes near the root of the clock tree have the largest capacitance of any nodes in the processor design and have the highest activity factors.

Exploit Locality to Reduce Data and Instruction Access Energy

We can exploit locality in instruction and operand accesses to improve energy efficiency. Typically, this entails using register organizations that keep operands close to function units, and using memory hierarchies that exploit reuse and locality to capture important instruction and data working sets close to processors. These organizations usually use small structures to reduce access energies and distribute storage structures close to function units to reduce communication and transport energies.

Amortize Instruction Fetch and Control

We can use single-instruction multiple-data (SIMD) architectures to reduce instruction bandwidth when applications exhibit regular data-level parallelism. Traditional vector processors improve efficiency by amortizing instruction fetch and decode energy by applying an instruction to multiple operands [59]. Many modern architectures provide collections of sub-word SIMD instructions that apply an operation to multiple operands that are packed in a common register. These schemes also eliminate or amortize the dependency checks that are required to determine when an operation may be initiated safely, which effectively reduces demand for control hardware bandwidth.

Use Simple Control and Decode Hardware

Most commercial embedded processors use simple hardware architectures to improve area and power efficiency. Many use RISC architectures that offer simple control and decode hardware. High-end digital signal processors often use VLIW architecture to efficiently exploit instruction-level parallelism found in many traditional signal processing algorithms [43]. These systems rely on compiler technology rather than hardware to discover and exploit instruction-level parallelism, and are able to scale to very wide issue. Many rely on predication to conditionally execute instructions to expose sufficient instruction-level parallelism. Traditional VLIW processors suffer from poor instruction density and large code sizes. Digital signal processors that use VLIW architectures rely on sophisticated instruction compression and packing schemes to eliminate the need for explicit no-ops and improve code sizes.

Exploit Operation-Level Parallelism with Compound Instructions

We can reduce the number of instructions that need to be executed to implement a computation by exploiting operation-level parallelism and encoding multiple simple dependent operations as a single machine instruction. For example, many modern architectures provide fused multiple-add instructions that encode a multiplication operation and a dependent addition operation. Like the instruction bundles used to encode independent instructions in VLIW architectures, compound instructions encode multiple arithmetic and logic operations; unlike instruction bundles, which encode multiple independent operations, compound instructions usually encode dependent operations.

Many architectures fuse short sequences of simple instructions into compound instructions to improve performance and reduce instruction bandwidth and energy. These can be common compound arithmetic operations that are provided by the hardware and selected by a compiler [105, 71]. They may be constructed by configuring stages in a function unit. For example, a processor could allow software to configure the stage that implements different rounding and saturating arithmetic mode[10]. They may rely on a combination of compiler analysis and hardware mechanisms for dynamically configuring a cluster of simple function units [23].

As with VLIW architectures, there is a limited number of useful operations that can be performed without additional register file bandwidth to deliver more operands to the compound instructions. The additional control logic, multiplexers, pipeline registers required by the additional function units, and the increased interconnect distances between the register file and function units that must be traversed increase the expense of executing simple instructions.

Exploit Application-Specific Instructions

We can provide instructions that implement highly stereotyped application-specific operations as sequences of dedicated instructions to reduce instruction and operand fetch bandwidth. Commercial digital signal processors often implement application-specific instructions to accelerate core kernels in important signal processing algorithms. For example, some digital signal processors for mobile applications provide

instructions that perform domain-specific operations, such as compare-select-max-and-store and square-distance-and-accumulate [140], to accelerate specific baseband processing algorithms in cellular telephone applications. Extensible processor architectures such as the Xtensa processor cores designed by Tensilica allow custom application-specific instructions to be introduced to a load-store instruction set architecture at design time [53]. A more aggressive example, recent TriMedia media processors implement complex super-instructions that issue across multiple VLIW instruction slots and function units [143]. The super-instructions implement specific operation that are used in the context-based adaptive binary arithmetic coding scheme used in advanced digital video compression standards. These super-instructions essentially embedded application-specific fixed function hardware in the function units.

The complex behaviors encoded by application-specific instructions can make it difficult for compilers to use application-specific instructions effectively, as the compiler must be designed to expose and identify clusters of operations that can be mapped to application-specific instructions. The problem is more acute when applications-specific instructions use unconventional data types and operands widths. Consequently, important application codes that can benefit from application-specific instructions often must be manually mapped to assembly instructions.

Switch to Multicore Architectures

Multicore architectures integrate multiple processor cores in a system-on-chip [112]. As microprocessor designs became increasingly more complex, using the additional transistors provided by technology scaling to implement multiple simple cores required considerably less engineering effort than attempting to design, implement, and verify increasingly more aggressive and complex speculative superscalar out-of-order processors [56]. Multicore architectures consciously prefer improved aggregate system performance to improved single-thread performance [5], and most multicore processors use increasing transistor budgets to exploit thread-level parallelism rather than instruction-level parallelism [15]. Because power dissipation constraints and wire delays limit the performance and efficiency of complex processors, almost all processor designs have shifted to a paradigm in which new processors provide more cores [127, 125, 144, 92, 8, 124], deliver modest microarchitecture improvements to increase single-thread performance and power efficiency [144, 92], and support additional thread-level parallelism within cores [97, 108]. To further improve energy efficiency, contemporary designs implement dynamic voltage and frequency scaling schemes [21] that allow single thread performance to be reduced in exchange for greater energy efficiency [55, 92]

Exploit Voltage Scaling and Parallelism

We can exploit voltage scaling to improve energy efficiency in applications with task-level and data-level parallelism by mapping parallel computation across multiple processors [153, 22]. The basic idea is to use parallelism to exploit the quadratic dependence of energy on supply voltage in digital circuits [21]. If we ignore the overheads of task and data distribution and communication, mapping parallel computation across multiple processors results in a linear increase in area and switched capacitance. In exchange, we can lower the supply voltage to produce a quadratic reduction in the dynamic energy per operation. As we should

expect, there are limits. Physical design considerations and constraints such as power supply noise and device variation often limit the extent to which supply voltages can be lowered reliably. Many circuits, such as conventional static random access memory cells, have a minimum supply voltage below which they will not operate reliably. Eventually, minimum supply voltage constraints, decreasing efficiency gains, or increasing circuit delays that may not be acceptable in latency-sensitive systems limit the extent to which voltage scaling can be used to improve efficiency. Furthermore, most consumer applications are cost sensitive, and the reduction in clock frequency and accompanying degradation in area efficiency result in what are often unacceptable increases in silicon and packaging costs.

Devise Manycore Architectures

The terms *manycore* and *massively multicore* are often used to refer to multicore architectures with tens to hundreds of cores. This definition is somewhat problematic because process scaling allows the number of cores in a multicore architecture to increase steadily. Multicore effectively describes architectures descended from single processors through processes in which the number of standard processor cores per die essentially doubles when a new semiconductor process generation doubles transistor densities [11]. Perhaps the term *manycore* distinguishes scalable architectures designed specifically to improve application performance and efficiency by distributing parallel computations across large numbers of highly integrated cores. These architectures often rely on explicitly parallel programming models to expose sufficient parallelism.

Mancycore architectures are specifically designed to exploit data-level and task-level parallelism within applications. They focus on delivering massive numbers of function units efficiently, and support massive amounts of parallelism. Unlike the communication primitives found in multicore processors, which descend from architectures designed to support multi-chip multiprocessors typically comprising from 2 to 16 processors, manycore architectures integrate scalable communication systems that exploit the low communication latencies and abundant communication bandwidth that are available on chip. Consequently, manycore architectures deliver significantly more arithmetic bandwidth than multicore processors, and usually achieve superior area and energy efficiency when handling applications with sufficient structure and locality. However, the ability of tiled manycore architectures that use conventional RISC microprocessors and digital signal processors [139, 52, 14, 46, 54, 153] to improve efficiency is limited by the efficiency of the processors from which they are constructed: tiling provides additional function units, which increases throughput, but does not improve the architectural efficiency of the individual cores.

2.3 A Survey of Related Contemporary Architectures

Data parallel architectures such as the Stanford Imagine processor [121], the MIT Scale processor [88], and NVIDIA graphics processors [99] amortize instruction issue and control overhead by issuing the same instruction to multiple parallel function units. However, the efficiency of data parallel architectures deteriorates when applications lack sufficient regular data parallelism. Sub-word SIMD architectures that use mechanisms such as predicate masks to handle divergent control flow [106, 98, 128] perform particularly poorly

when applications lack adequate regular data parallelism because multiple control paths must be executed and operations discarded [135, 73]. The following sections discuss in detail several architectures that are representative of alternative efficient embedded and application-specific processor architectures.

MIT Raw and Tilera TILE Processors

The MIT Raw processor is perhaps the canonical example of a tiled multicore processor. The Raw processor comprises 16 simple RISC cores connected by multiple mesh networks [137]. The Raw architecture exposes low-level computation, storage, and communication resources to software, and relies on compiler technology to efficiently map computations onto hardware [147]. The Raw architecture exposes software-scheduled scalar operand networks that are tightly coupled to the function units for routing operands between tiles [138]. The Raw architecture was designed to exploit instruction-level parallelism; its architects designed the architecture to scale beyond what were considered to be communication limitations of conventional superscalar architectures by exposing wire delays and distributed resources directly to software [139]. The Raw compiler models the machine as a distributed register architecture and attempts to map the preponderance of communication between function units onto these scalar operand networks using compiler technologies such as space-time scheduling of instruction-level parallelism [96]. The Raw architecture was better able to exploit task-level and data-level parallelism than instruction-level parallelism, and could be effectively programmed using stream programming languages such as StreamIt [141, 75, 49].

The Tilera TILE architecture is the commercial successor to the MIT Raw architecture. The TILE64 processor is a system-on-chip with 64 conventional 3-wide VLIW cores that are connected by multiple mesh networks [18]. The TILE processor architecture adds support for sub-word SIMD operations, which improve the performance of various video processing applications that process 8-bit and 16-bit data. The TILE architecture provides both software-routed scalar operand networks and general-purpose dynamically routed networks, and retains the ability to send data directly from registers and receive data in registers when communication is mapped to the software-controlled networks [148]. To decouple the order in which messages are received from the order in which software processes the messages, the software-controlled networks implement multiple queues and allow messages to be tagged for delivery to a specific queue. A shared SRAM stores messages at the receiver, and small dedicated per-flow hardware queues provide direct connections to the function units [18]. The scalar operand networks improve the efficiency at which fine-grain pipeline-parallelism and task-parallelism can be mapped to the architecture.

Berkeley VIRAM Vector Processor

The Berkeley VIRAM processor is a vector processor that uses embedded DRAM technology to implement a processor-in-memory. Like conventional vector processors, VIRAM has a general-purpose scalar control processor and a vector co-processor. It implements 4 64-bit vector lanes and a hardware vector length of 32 [84]. The architecture allows instructions to operate on sub-word SIMD operands within each lane, which increases the effective vector length of the machine when operating on packed data elements. When sufficient regular data-level parallelism is available in a loop, increasing the vector length reduces instruction

bandwidth, as a single vector instruction is applied to more elements. To allow data-parallel loops that contain control-flow statements to be mapped to the vector units, the VIRAM instruction set architecture includes vector flags, compress, and expand operations to support conditional execution.

Stanford Imagine and SPI Storm Stream Processors

Stream processors are optimized to exploit the regular data-level parallelism found in signal processing and multimedia applications [81, 74, 35]. Like vector processors, stream processor architectures have a scalar control processor that issues commands to a co-processor comprising a collection of data-parallel execution units. Stream applications are expressed as computation kernels that operate on streams of data. The kernels are compute-intensive functions that operate on data streams, and usually correspond to the critical loops in an application. The data streams are sequences of records. The data stream construct makes communication between kernels explicit, which greatly simplifies the compiler analysis that are required to perform code transformations that improve data reuse and locality. The control processor coordinates the movement of data between memory and the data-parallel execution units, and launches kernels that execute on the data-parallel execution units. Stream memory operations transfer data between memory and a large stream register file that stages data to and from the data-parallel units. Stream processors differ most significantly from conventional vector architectures in their use of software-managed memory hierarchies to allow software to explicitly control the staging of stream transfers through the memory hierarchy [121]. The storage hierarchy allows intra-kernel reuse and locality to be captured in registers, and inter-kernel locality to be captured in a large on-chip stream register file. The stream register file is also used to stage large bulk data transfers between DRAM and registers. The capacity of the stream register file allows it to be used to tolerate long memory access times. Irregular data-level parallelism requires the use of techniques such as conditional streams to allow arbitrary stream expansion and stream compression in space and time [73].

The Stanford Imagine processor has 8 data-parallel lanes, each of which has 7 function units [81]. Imagine uses a micro-sequencer to coordinate the sequencing of kernel invocations and issuing of the instructions comprising a kernel to the data-parallel units, which decouples the launching of a kernel on the control processor from the execution of the data-parallel units. This differs somewhat from conventional vector processors that issue individual vector instructions in order to the vector unit [10]. The function units within a lane allow Imagine to exploit instruction-level parallelism, and the multiple lanes allow Imagine to exploit SIMD data-level parallelism. The SPI Storm-1 processor is the commercial successor to Imagine. It integrates the control processor on the same chip as the data-parallel execution units, and increases the number of data-parallel lanes from 8 to 16 [80].

MIT Scale Processor

The MIT Scale vector-thread processor supports both vector and multithreaded processing. It implements a vector-thread architecture in which a control thread executes on a control processor and issues instructions to a collection of micro-threads that execute concurrently on parallel vector lanes [88]. The control thread allows common control and memory operations to be factored out of the instruction streams that are issued

to the micro-threads, which allows regular control-flow and data accesses to be handled efficiently. The control processor issues vector fetch instruction to dispatch blocks of instructions to the micro-threads. The architecture allows micro-threads to execute independent instruction streams, which improves the handling of irregular data-parallel applications. For example, the micro-threads may diverge after executing scalar branches contained in a block of instructions that were dispatched from a vector fetch command. To support divergent micro-thread execution, the Scale processor provisions more instruction fetch bandwidth than a conventional vector processor would. The Scale processor implements a clustered function unit organization that supports 4-wide VLIW execution within each vector lane [87], which allows it to exploit well structured instruction-level parallelism. To reduce the amount of hardware and miss state required to sustain the large number of in-flight memory requests that are generated, Scale implements a vector refill unit that pre-executes vector memory operations and issues any needed cache line refills ahead of regular execution [17].

Illinois Rigel Accelerator

The Illinois Rigel Accelerator is a programmable accelerator architecture that emphasizes support for data-parallel and task-parallel computation [77]. It provides a collection of simple cores and hierarchical caches, and emphasizes support for single-program multiple-data models of computation. The memory hierarchy has two levels: a local cluster cache that is shared by 8 processors, and a distributed collection of global cache banks that are associated with off-chip memory interfaces. To simplify the mapping of applications to the accelerator, the programming model structures computation as collections of parallel tasks. Interprocessor communication uses shared memory and is implicit. The programming system provides a cached global address space, and relies on a software memory coherence scheme to reduce communication overhead. Explicit synchronization instructions are provided to allow the runtime system to implement barriers, and to accelerate the enqueuing and dequeuing tasks from distributed work queues. The instruction set architecture exposes mechanisms that support implicit locality management, such as load and store instructions that specify where data may be cached. The memory consistency model provides coherence at a limited number of well-defined points within an application, and relies on software to enforce the cache coherence protocol [78].

2.4 Chapter Summary

This chapter presented an analysis of efficiency of a simple embedded RISC processor that provides a reasonable approximation of an efficient modern embedded processor. The analysis reveals that the preponderance of the energy consumed in the processor is used to transfer instructions and data to function units, not to compute. This result explains and quantifies the significant disparities between the efficiencies of programmable processors and application-specific fixed-function hardware. It also implies a limit to the efficiencies that can be achieved using conventional processor architectures in manycore systems. Mapping parallel computations to collections microprocessors and digital signal processors provides additional function units, which improves throughput, but does not improve the architectural efficiencies of the cores. As a consequence, aggregate system efficiency will ultimately be limited by the efficiency of the individual processor cores.

Chapter 3

The Elm Architecture

This chapter introduces the Elm architecture. It motivates and develops important concepts that informed the design of Elm, and presents many of the ideas that are developed in subsequent chapters.

3.1 Concepts

Embedded applications have abundant task-level and data-level parallelism, and there is often considerable operation-level parallelism within tasks. Components of embedded systems such as application-specific integrated circuits exploit this abundant parallelism by distributing computation over many independent function units; both performance and efficiency improve because instructions and data are distributed close to function units. Efficiency is further improved by exploiting extensive reuse and producer-consumer locality in embedded applications. Individual components in embedded systems tend to perform the same computation repeatedly, which is why they can benefit from fixed-function logic. This behavior may be used to expose extensive instruction and data reuse.

Programmability allows tasks to be time-multiplexed onto collections of function units, which both improves area efficiency and energy efficiency. Systems that use special-purpose fixed-function logic usually provision dedicated hardware for each application and tasks. Depending on application demand, some fixed-function hardware will be idle, resulting in resource underutilization and poor area efficiency. For example, cellular telephone handsets that support multiple wireless protocols typically implement multiple baseband processing blocks even though multiple blocks cannot be used concurrently. Time multiplexing tasks onto programmable hardware allows a collection of function units to implement different applications and different tasks within applications as demanded by the system. Programmability improves energy efficiency when data communication dominates by allowing tasks to be dispatched to data. Rather than requiring that data be transferred to and from hardware capable of implementing some specific task, the task may be transferred to the data, reducing data communication and aggregate instruction and data bandwidth demand.

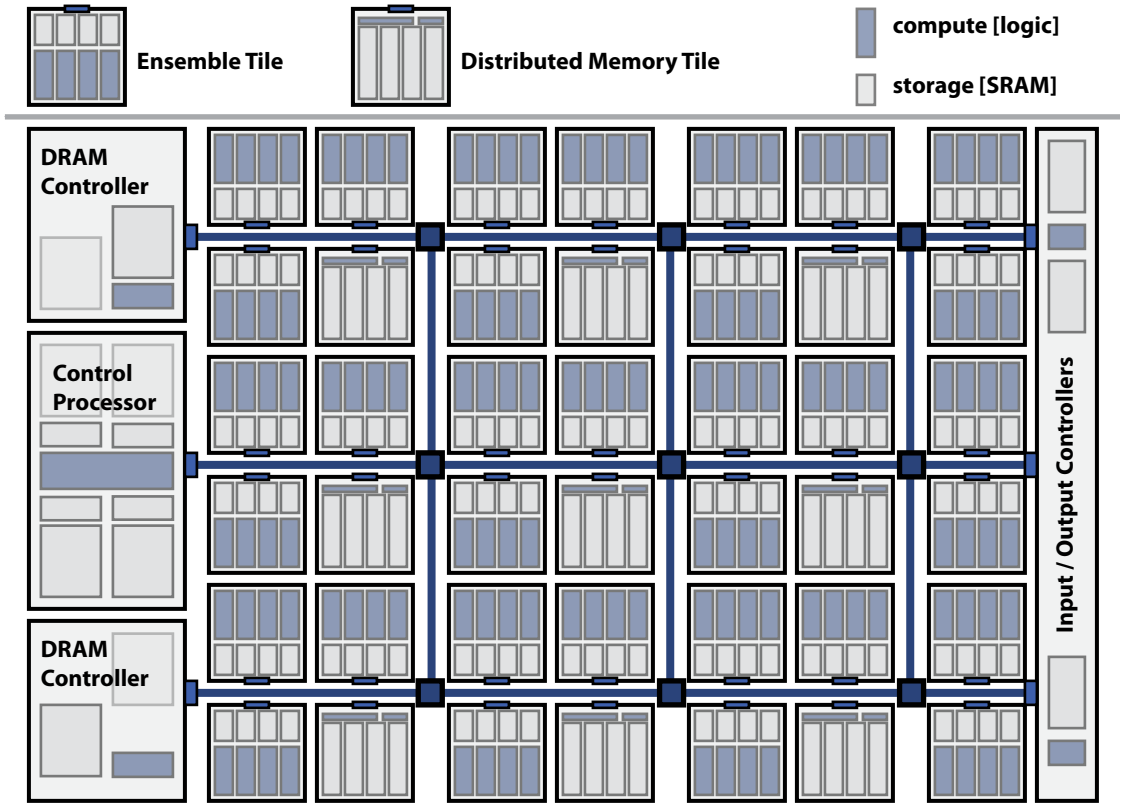


Figure 3.1 – **Elm System Architecture.** The system comprises a collection of processors, distributed memories, memory controllers, and input-output controllers. The modules are connected by an on-chip interconnection network.

3.2 System Architecture

The Elm architecture exploits the abundant parallelism, reuse, and locality in embedded applications to achieve performance and efficiency. Figure 3.1 provides a conceptual illustration of an Elm system-on-chip. The latency-optimized control processor handles system management tasks, while the computationally demanding parts of embedded applications are mapped to the dense fabric of efficient Elm processors. Efficiency is achieved by keeping a significant fraction of instruction and data bandwidth close to the function units, where hierarchies of distributed register files and local memories deliver operands and instructions to the function units efficiently. The processor complexity is low enough that custom circuit design techniques could be used throughout to improve performance and efficiency, and small enough for large amounts of arithmetic bandwidth to be implemented economically. We estimate that an Elm system implemented in a 45 nm CMOS process would deliver in excess of 500 GOPS at less than 5 W in about 10 mm $\times$ 10 mm of die area.

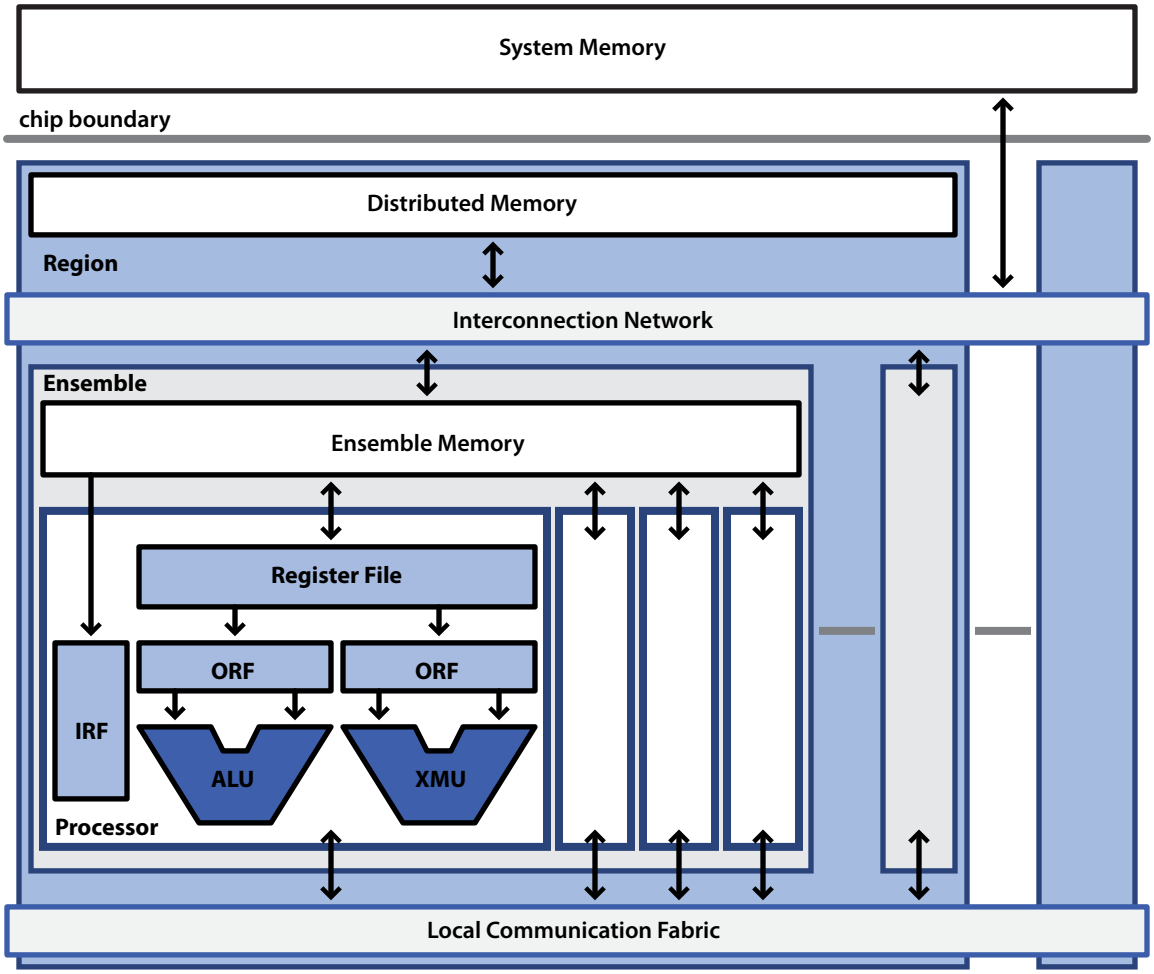


Figure 3.2 – **Elm Communication and Storage Hierarchy.** Instructions are issued from instruction register files (IRFs) distributed among the function units. Distributed operand register files (ORFs) provide inexpensive locations for capturing short-term reuse and locality close to function units. Collections of processors are grouped to form Ensembles, which share a local Ensemble Memory. Collections of Ensembles are grouped with a distributed memory tile to form a region. The local communication fabric provides low-latency, high-bandwidth communication links and allows processors to communicate through registers.

Figure 3.2 illustrates the distributed and hierarchical register and memory organization. The Elm processors are statically scheduled dual-issue 32-bit processors. Instructions are issued in pairs, with one instruction sent to the ALU pipeline and one to the XMU pipeline. The ALU executes complex arithmetic, logic, and shift instruction; the XMU executes simple arithmetic, memory, control, communication, and synchronization instructions. Instructions are issued from registers to reduce the cost of delivering instructions. The distributed and hierarchical register organization reduces the cost of accessing local instructions and data.

The Elm processors use block instruction and data transfers to improve the efficiency of transferring instructions and data between registers and memory.

The processors communicate over an on-chip interconnection network and a distributed local communication fabric. The memory and communication systems near the processors provide predictable latencies and bandwidth to simplify the process of compiling to real-time performance and efficiency constraints. The compute, memory, and communication resources are exposed to software to provide fine-grain control over how computation unfolds, and the programming systems and compiler may explicitly orchestrate instruction and data movement. The result is that the complexity and expense of developing large embedded systems is in the development of compilers and programming tools, the cost of which is amortized over many systems, rather than in the design and verification of special-purpose fixed-function hardware which must be paid for in each new system.

Ensembles

Clusters of four processors are grouped into Ensembles, the primary physical design unit for compute resources in an Elm system. The processors in an Ensemble are loosely coupled, and share local resources such as a pool software-managed memory and an interface to the interconnection network. The Ensemble organization is exposed to software, and the software concept of an Ensemble provides a metaphor for describing and encoding spatial locality among the processors within an Ensemble.

The Ensemble memory captures instruction and data working sets close to the processors, and is banked to allow the local processors and network interface controller to access it concurrently. The compiler can exploit task-level parallelism by executing parallel tasks on the processors in an Ensemble, capturing fine-grain producer-consumer data locality within the Ensemble to improve efficiency. Reuse among the processors reduces the demand for instruction and data bandwidth outside the Ensemble. For example, executing parallel iterations of a loop on different processors in an Ensemble increases performance by exploiting data-level parallelism; amortizing the cost of moving the instruction and data to Ensemble memory increases efficiency by exploiting instruction and data reuse across iterations.

Remote memory accesses are performed over the on-chip interconnection network. The Ensemble memory can be used to stage transfers. Hardware at the network interface supports bulk memory operations to assist the compiler in scheduling proactive instruction and data movements to hide remote memory access latencies. The Ensemble organization effectively concentrates interconnection network traffic at Ensemble boundaries and allows multiple processors share a network interface and the associated connection to the local router. This amortizes the network interface and network connection across multiple processors, which provides an important improvement in area efficiency because the processors are relatively small.

The local communication fabric connects the processors in an Ensemble with high-bandwidth point-to-point links. The fabric can be used to capture predictable communication, and allows the compiler to map instruction sequences across multiple processors. The fabric can also be used to stream data between Ensembles deterministically at low latencies. The fabric is statically routed, and avoids the expense cycling large memory arrays incurs when communicating through memory.

Efficient Instruction Delivery

Elm uses compiler-managed instruction registers (IRs) to store instructions close to function units. Instructions are issued directly from instruction registers. Instruction registers keep a significant fraction of the instruction bandwidth close to the function units by capturing reuse and locality within loops and kernels. The instruction registers are organized as shallow instruction register files (IRFs) that are located close to the function units. The register files are fast and inexpensive to access, and their proximity to the function units reduces the energy required to transfer instructions to the datapaths. The shallow register files require few bits to address, allowing a short instruction counter (IC) to be used instead of a conventional program counter. An IC is less expensive to maintain, which reduces the energy consumed coordinating instruction issue and transfers of control. The contents of the instruction registers and the instruction counter implicitly define the state captured by a program counter. A complete set of control-flow instructions is implemented, and arbitrary transfers of control are permitted, provided that the target instruction resides in a register. Relative target addresses are resolved while instructions are loaded to improve efficiency: the hardware that resolves relative targets is active only when instructions are being loaded.

The instruction registers are backed by the Ensemble memory. The compiler explicitly orchestrates the movement of instructions into registers by scheduling instruction loads. An instruction load moves a contiguous block of instructions from memory into registers. The compiler specifies the number of instructions to be loaded, where the instructions reside in memory, and into which registers instructions are to be loaded. A status bit associated with each instruction register records whether a load is pending. The processor continues to execute during instruction loads, stalling when it encounters a register with a load pending. Decoupling the loading and issuing of instructions allows the compiler to hide memory access latencies when loading instructions by anticipating control transfers and proactively scheduling loads. The instruction registers reduce the demand for instruction bandwidth at the Ensemble memory by filtering a significant fraction of the instruction references, which allows a single memory read port and a single memory write port to be used to access both instructions and data. This simplifies the memory arbitration hardware, and reduces the degree of banking required in the Ensemble memory.

Instructions can be loaded directly from remote and system memory, avoiding the cost of staging instructions in the local Ensemble memory, and the distributed memories located throughout the system can be used as hardware-managed instruction caches. The caches assist software by providing a mechanism for capturing instruction reuse across kernels and control transfers that are difficult for the compiler to discover.

Scheduling the compiler-managed instruction registers requires additional compiler analysis, allocation, and scheduling passes, which increase the complexity of the instruction scheduler implemented in the Elm compiler. In exchange, the instruction register organization provides better efficiency and performance than an instruction cache. Because they do not require tags, the instruction registers are faster and less expensive to access than a cache of the same capacity. The tag overhead in a small cache with few blocks is significant because the tag needs to cover most of the address bits, as few of the address bits are covered by the index and block offset. Although increasing the block size will reduce the tag overhead, doing so increases conflict

misses because there are fewer entries. It also leads to increased fragmentation within the blocks, increasing the probability that instructions will be unnecessarily transferred to the cache. The instruction register organization allows fragmentation to be avoided by specifying transfers at instruction boundaries rather than cache block boundaries.

Because the compiler has complete control over where instructions are placed in the IRF and when instructions are evicted, the compiler is able to reduce misses and is able to schedule instruction loads well in advance of when the instructions will be issued to avoid stalls in the issue stage of the pipeline. Conversely, software has less control over where instructions are placed in an instruction cache, which makes it more difficult to determine which instructions will experience conflict misses unless the cache is direct-mapped. This becomes significant when mapping the kernels that dominate application performance to small instruction stores. Furthermore, most instruction caches load instructions in response to a miss, which causes the processor to stall each time control transfers to an instruction that is not in the cache. Software-initiated instruction prefetching allows some cache miss latencies to be hidden but suffers two disadvantages: the compiler has little or no control over which instructions are evicted, and instruction prefetches are typically limited to transferring a single cache block. Many prefetch instructions must be executed to transfer what a single instruction register load would. Hardware prefetch mechanisms reduce the need to explicitly prefetch instructions, but cannot anticipate complicated transfers of control. Most simply prefetch ahead in the instruction stream based on the current program counter or the address of recent misses, and may transfer instructions that will not be executed to the cache.

Efficient Operand Delivery

Elm uses a distributed and hierarchical register organization to exploit reuse and locality so that a significant fraction of operands are delivered from inexpensive registers that are close to the function units. The register organization provides software with extensive control over the placement and movement of operands. Shallow operand register files (ORFs) located at the inputs of the function units capture short-term data locality and keep a significant fraction of the data bandwidth local to each function unit. The shallow register files are fast and inexpensive to access. Placing the register files at the inputs to the function units reduces the energy used to transfer operands and results between registers and the function units. Operands in an ORF are read in the same cycle in which the operation that consumes them executes; this simplifies the bypass network and reduces the number of operands that pass through pipeline registers. The distributed ORF locations are exposed to allow software to place operands close to the function units that will consume them. The pipeline registers that receive the results of the function units are exposed as result registers, which can be used to explicitly forward values to subsequent instructions. A result register always holds the result of the last instruction executed on the function unit. The compiler explicitly forwards ephemeral values through result registers to avoid writing dead values back to the register files, which can consume as much energy as simple arithmetic and logic operations. Explicit operand forwarding also keeps ephemeral values from unnecessarily consuming registers, which relieves register pressure and avoids spills.

The general-purpose register file (GRF) and larger indexed register file (XRF) form the second level of the

register hierarchy. The capacity of the GRF and XRF allow them to capture larger working sets, and to exploit data locality over longer intervals than can be captured in the first level of the register hierarchy. The general-purpose registers back the operand registers. Reference filtering in the first level of the hierarchy reduces demand for operand bandwidth at the GRF and XRF, allowing the number of read ports to the GRF and XRF to be reduced. Registers in the second level of the hierarchy are more expensive to access because the register files are larger and further from the function units, but the additional capacity allows the compiler to avoid more expensive memory accesses. Because both the ALU and XMU can read the registers in the GRF, the compiler may allocate variables that are used in both function units to the GRF rather than replicating variables in the first level of the register hierarchy.

Bulk Memory Operations

Elm allows the XRF to be accessed indirectly through index pointer registers (XPs), which are located in the index-pointer register file (XPF). This allows the XRF to capture structured reuse within arrays, vectors, and streams of data, structured reuse that would otherwise be captured in significantly more expensive local memory. Each XP provides a read pointer and a write pointer. The read pointer is used to address the XRF when the XP appears as an operand, and the write pointer is used to address the XRF when the XP appears as a destination. Hardware automatically updates pointers after each use so that the next use of the XP register accesses the next element of the array, vector, or stream. Software controls how many registers in the XRF are assigned to each XP by specifying a range for each XP, and hardware automatically wraps pointers after each update so that they always points to an element in their assigned range.

Elm provides a variety of bulk memory operations to allow instructions and data to be move efficiently through the memory system. To reduce the number of instructions that must be executed to move data between memory and the register hierarchy, Elm provides high-bandwidth vector load and store operations. Once initiated, vector memory operations complete asynchronously, which decouples the execution of vector loads and stores from the access of the memory port when moving data between registers and memory. For example, the XMU may execute arithmetic operations while a vector memory operation completes in the background. Vector memory operations may use the XRF to stage transfers between memory and registers. By automating address calculations, vector memory operations also reduce the number of instructions that must be executed to calculate an address before executing a load or store. When used to move data between the XRF and memory, vector memory operations provide twice the bandwidth of their scalar equivalents, transferring up to 64-bits each cycle. Vector gather and scatter operations can be implemented using the indexed registers to deliver indices. To allow software to schedule the movement of individual elements while exploiting the reduced overheads of vector memory operations, Elm exposes most of the hardware used to implement vector memory operations to software. This allows software to transfer elements directly between operand registers and memory, avoiding the expense of using the large indexed register files to stage data.

Local Communication

Elm allows processors to communicate through registers using the local communication fabric. Dedicated message registers are used to send and receive data over the local communication fabric. The message registers provide single-cycle transport latencies within an Ensemble: operands arrive in the cycle after being sent and can be consumed immediately. Synchronized send and receive operations allow code executing on different processors to explicitly synchronize on message registers. The synchronized communication instructions use presence bits associated with each message register to track whether a local message register is empty, and whether a remote message register is full. Non-blocking send and receive operations are provided to transfer values efficiently between processors whose execution is synchronized. The non-blocking communication instructions sustain higher transfer bandwidths because the sender does not need to wait for the receiver to acknowledge having accepted a value, and because compiler can schedule predicated send operations between processors as it would conditional move operations within a processor.

Processors Corps

The Ensemble organization reduces the cost of delivering instructions when data-parallel codes are executed across the processors within an Ensemble by capturing shared instruction working sets in the Ensemble memory. For example, when executing a parallel loop across an Ensemble, a single bulk transfer may be used to load the loop's instructions into the Ensemble memory before distributing them to the instruction registers. This amortizes the energy expended transferring instructions to the Ensemble, and reduces demand for instruction bandwidth outside the Ensemble.

Single-instruction multiple-data (SIMD) parallelism can be exploited to reduce further the cost of delivering instructions within an Ensemble. SIMD parallelism is exploited within an Ensemble using the concept of processor corps. Processors operating as a processor corps execute the same instruction stream: one processor in the corps fetches instructions from its instruction registers and broadcasts them to the others in the group. Software explicitly orchestrates the creation and destruction of processor corps. Instructions may be issued from any of the instruction registers belonging to the processors participating in a processor corps. The additional instruction register capacity improves efficiency when larger instruction working sets are captured in registers, as fewer instructions are loaded from memory. Distributing instructions to multiple processors improves energy efficiency by amortizing the cost of reading the instruction register files, although the energy expended transferring instructions to the datapath control points increases because instructions must traverse greater distances.

Processors can quickly form and leave processor corps. Processors form a processor corps by executing a distinguished jump instruction that allows control to transfer to any of the instruction registers in an Ensemble. Processors leave a processor corps by transferring control back into their local instruction registers. The processors wait until all members have joined, which creates an implicit synchronization event. This allows the compiler to use processor corps to schedule parallel iterations of loops and independent invocations of a kernel across multiple processors efficiently even when the code contains conditional statements.

To simplify the mapping of data parallel computations to processor corps, the address generation step

of the standard load and store instruction uses the local address registers to generate the effective memory address. Each processor within the group generates its own effective address, which allows the processors to access arbitrary memory locations. Additional load and store instructions that insert the processor identifier into the bank selection bits of the effective address are available for accessing data that has been stripped across the banks, which ensures that each processor accesses a different bank. Elm provides additional vector load and store instructions in which the effective address computed by one processor is forwarded to multiple memory banks to eliminate unnecessary address computations when accessing contiguous memory locations in a processor corps. Additional message register names are available to processors in an issue group to simplify the naming of other processors within the group; the additional register names are aliases for existing physical message registers. The predicate registers are partitioned into group and local registers. Group predicate registers are kept consistent across an issue group by broadcasting updates to all of the processors in the group. This allows predicated instructions to execute consistently across the issue group. Local predicate registers are private to each processor, and updates are not visible outside the local processor. Software can use the local predicate registers to control which processors execute instructions that are issued to a group, or to hold predicates that are used when executing outside of an issue group.

Distributed Stream and Cache Memory

Elm is designed to support many concurrent threads executing across an extensible fabric of processors. To improve performance and efficiency, Elm implements a hierarchical and distributed on-chip memory organization in which the local Ensemble memories are backed by a collection of distributed memory tiles. Elm exposes the memory hierarchy to software, and allows software to control the placement and communication of data explicitly. This allows software to exploit the abundant instruction and data reuse and locality present in embedded applications. Elm allows software to transfer data directly between Ensembles using Ensemble memories at the sender and receiver to stage data. This allows software to be mapped to the system so that the majority of memory references are satisfied by the local Ensemble memories, which keeps a significant fraction of the memory bandwidth local to the Ensembles. Elm provides various mechanisms that assist software in managing reuse and locality, and that assist software in scheduling and orchestrating communication. This also allows software to improve performance and efficiency by scheduling explicit data and instruction movement in parallel with computation.

3.3 Programming and Compilation

Elm relies extensively on programming systems and compiler technology, some of which is described in this dissertation. The instruction set architecture allows effective optimizing compilers to be constructed using existing compiler technology. The effectiveness of these optimizing compilers can be improved by incorporating various compiler technologies we developed during the design the Elm architecture and the implementation of Elm. This section describes various software tools that we developed for Elm.

```
1  actor FIR<int N> {
2      // [ local state ]
3      const int[N] coeffs;
4
5      // [ constructor ]
6      FIR(const T[N] coeffs) {
7        this.coeffs = coeffs;
8      }
9
10     // [ stream map operator ]
11     (int[N] in stride=1) -> (int[1] out) {
12       int sum = 0;
13       for (int i = 0; i < N; i++) {
14         sum += coeffs[i]*in[i];
15       }
16       out[0] = sum;
17     }
18   }
19 }
```

Figure 3.3 – **Elk Actor Concept.** The code fragment shown implements a finite impulse response filter.

The Elk Programming Language

Elk is a parallel programming language for mapping embedded applications to parallel manycore architectures. The Elk language is based on a concept for a parallel programming language I designed for Elm; the language allows structured communication to be exposed as operations on data streams, and allows real-time constraints to be expressed as annotations that are applied to streams. Jongsoo Park was responsible for the principle design and implementation of the Elk language. Aspects of the Elk programming language syntax that he devised were inspired by the StreamIt language [141].

The Elk compiler and run-time systems were developed to allow applications to be mapped to Elm systems. The Elk programming language allows applications developers to convey information about the parallelism and communication in applications, which allows the compiler to automatically parallelize applications and implement efficient communication schemes.

The Elm elmcc Compiler Tools

The elmcc compiler is an optimizing C and C++ compiler that we developed to compile kernels. It may be used to compile applications, though programmers must manually partition applications and explicitly map parallel kernels and threads to processors. The elmcc compiler produces load position independent code for a single thread that will execute correctly on any arbitrary processor within the system. The compiler recognizes various intrinsic functions that provide low-level access to hardware mechanisms, such as the local communication fabric and stream memory operations. The primary purpose of the intrinsic functions is to provide an application programming interface for high-level programming systems and run-times such

```
1  bundle BandPassFilter<int N> {
2    int[N] hpfCoeffs, lpfCoeffs;
3
4    BandPassFilter(int lowFreq, int highFreq) {
5      computeHpfCoeffs(hpfCoeffs, lowFreq);
6      computeLpfCoeffs(lpfCoeffs, highFreq);
7    }
8    static void computeHpfCoeffs(int[N] coeffs, int cutoff) {
9      ...
10   }
11   static void computeLpfCoeffs(int[N] coeffs, int cutoff) {
12     ...
13   }
14
15   // [ stream map operator ]
16   (int[] in) -> (int[] out) {
17     in >> Fir<N>(hpfCoeffs) >> Fir<N>(lpfCoeffs) >> out;
18   }
19 }
```

Figure 3.4 – **Elk Bundle Concept.** The code fragment shown implements a band-pass filter as a low-pass filter and high-pass filter.

```
1  bundle Main {
2    () -> () {
3      stream<int> in rate=1MHz;
4      FileReader("in.dat")
5          >> in
6          >> BandPassFilter<32>(1000, 3000)
7          >> FileWriter("out.dat");
8    }
9  }
```

Figure 3.5 – **Elk Stream Concept.** The code fragment shown defines a 1 MHz stream and passes the stream through a band-pass filter.

as those used to implement Elk to access low-level hardware mechanisms; they also allow programmers to program at a very low-level of abstraction without resorting to assembly.

The elmcc compiler uses a standard GNU C/C++ compiler front-end and a custom optimizing back-end. The compiler uses the LLVM [94] framework to pass compiled and optimized virtual machine code from the front-end to the back-end. The back-end maps the low-level virtual machine code to Elm machine code. The elmcc back-end implements a number of machine-specific optimizations, such as instruction register scheduling and allocation, local communication scheduling, and operand register allocation; it also implements standard instruction selection and scheduling optimizations. Implementing the elmcc compiler this way allows us to leverage the extensive collections of high-level code transformations and optimizations that have been integrated in the GNU C/C++ compiler. The elmcc compiler allows a machine description file

```
1  actor CombineDft<int N> {
2    complex_f[N/2] w; // coefficients
3    CombineDft() {
4      for (int i = 0; i < N/2; i++) {
5        w[i] = complex_f(cos(-2*PI*i/N), sin(-2*PI*i/N));
6    } }
7
8    (complex_f[N] in) -> (complex_f[N] out) {
9      for (int i = 0; i < N/2; i++) {
10       complex_f y0 = in[i];
11       complex_f y1 = in[N/2 + i];
12       complex_f y1w = y1*w[i];
13       out[i] = y0 + y1w;
14       out[N/2 + i] = y0 - y1w;
15 } } }
16
17 actor FftReorderSimple<int N> {
18   (complex_f[N] in) -> (complex_f[N] out) {
19     for (int i = 0; i < N; i+=2) {
20       out[i/2] = in[i];
21     }
22     for (int i = 1; i < N; i+=2) {
23       out[N/2 + i/2] = in[i];
24 } } }
25
26 bundle FftReorder<int N> {
27   (complex_f[] in) -> (complex_f[] out) {
28     stream<complex_f>[log2(N)] temp;
29     in >> temp[0];
30     for (int i = 1; i < (N/2); i*=2) {
31       temp[log2(i)] >> FftReorderSimple<N/i>() >> temp[log2(i) + 1];
32     }
33     temp[log2(N) - 1] >> out;
34 } }
35
36 bundle Main {
37   () -> () {
38     int N = 256;
39     stream<complex_f>[log2(N) + 1] temp;
40     FileReader("in.dat") >> FftReorder<N>() >> temp[0];
41     for (int i = 2; i <= N; i *= 2) {
42       temp[log2(i) - 1] >> CombineDft<i>() >> temp[log2(i)];
43     }
44     temp[log2(N)] >> FileWriter("out.dat", 3072);
45 } }
```

Figure 3.6 – Elk Implementation of FFT.

to be passed as a command line argument. Machine description files describe the capabilities and resources of the target machine, and are typically used to conduct sensitivity experiments in which machine resources such as the number of operand registers are varied.

The elmcc linker allows multiple object files produced by the elmcc compiler to be combined into Ensemble and application bundles. The linker resolves and binds those symbols and locations that can be resolved at link time, such as the locations of data and code objects in the system and Ensemble address spaces. Like the elmcc compiler, the linker allows a machine description file to be passed as a command line argument.

The elmcc loader is responsible for binding any remaining unresolved symbols when applications load. The loader requires both an application bundle and an application mapping file. The application bundle contains the application code and data objects, and may contain symbolic references to Ensembles, distributed memories, processors, and so forth. The application mapping file specifies where code and data objects should be placed when an application is loaded, and is used to resolve symbolic address and location references. The modified application bundle produced by the loader may contain a combination of load position independent code, which will execute correctly on any processor within a system; load position dependent code, which only executes correctly on a specific processor; and relative load position dependent code, which will execute correctly provided that other code objects are mapped similarly. Relative load position code typically contains references that access the local communication fabric or local Ensemble memory. The loader requires a machine description file that specifies the Ensemble and memory resources of a machine.

3.4 Chapter Summary

This chapter described an Elm system and introduced many of the concepts and ideas that will be developed in subsequent chapters. A common theme throughout the thesis will be ways in which we can improve efficiency by exposing distributed computation, communication, and storage resources to software. This allows software to exploit parallelism and locality in applications.

The following five chapters extend the concepts introduced in this chapter and describe the Elm architecture in greater detail. The chapters follow a common organization structure a common structure. We present the major concepts presented in the chapter. We describe the application of these concepts to Elm, using specific mechanisms in the Elm architecture to provide concrete examples of how general concepts, and then present examples that illustrate how the ideas presented in architecture section improve efficiency. We then present a quantitative evaluation of the concepts and architectural mechanisms presented in the chapter. The evaluation is followed by a summary of related work.

Chapter 4

Instruction Registers

This chapter introduces instruction registers, distributed collections of software-managed registers that extend the memory hierarchy used to deliver instructions to function units. Instruction registers exploit instruction reuse and locality to improve efficiency. They allow critical instruction working sets such as loops and kernels to be stored close to function units. Instruction registers are implemented efficiently as distributed collections of small instruction register files. Distributing the register files allows instructions to be stored close to function units, reducing the energy consumed transferring instructions to data-path control points. The aggregate instruction storage capacity provided by the distributed collection of register files allows them to captured important working sets, while the small individual register files are fast and inexpensive to access.

The remainder of this chapter is organized as follows. I explain general concepts and describe the operation of instruction registers. I then describe microarchitectural details of the instruction register organization implemented in Elm. This is followed by examples illustrating the use of instruction registers. I then describe compiler algorithms for allocating and scheduling instruction registers. I conclude this chapter with a comparison of the performance and efficiency of instruction registers to similar mechanisms for reducing the amount of energy expended delivering instructions.

4.1 Concepts

Embedded applications are dominated by kernels and loops that have small instruction working sets and extensive instruction reuse and locality. Consequently, a significant fraction of the instruction bandwidth demand in embedded applications can be delivered from relatively small memories.

The instruction caches and software-managed instruction memories found in most embedded processors have enough capacity to accommodate working sets that are larger than most embedded kernels. Often, first-level instruction caches are made as large as possible while still allowing the cache to be accessed in a single cycle at the desired clock frequency. Consequently, the instruction caches found in common embedded processors are much larger than a single kernel or loop body: it is not uncommon to find embedded processors with instruction caches ranging from 4 KB to 32 KB, while many of the important kernels and loops in

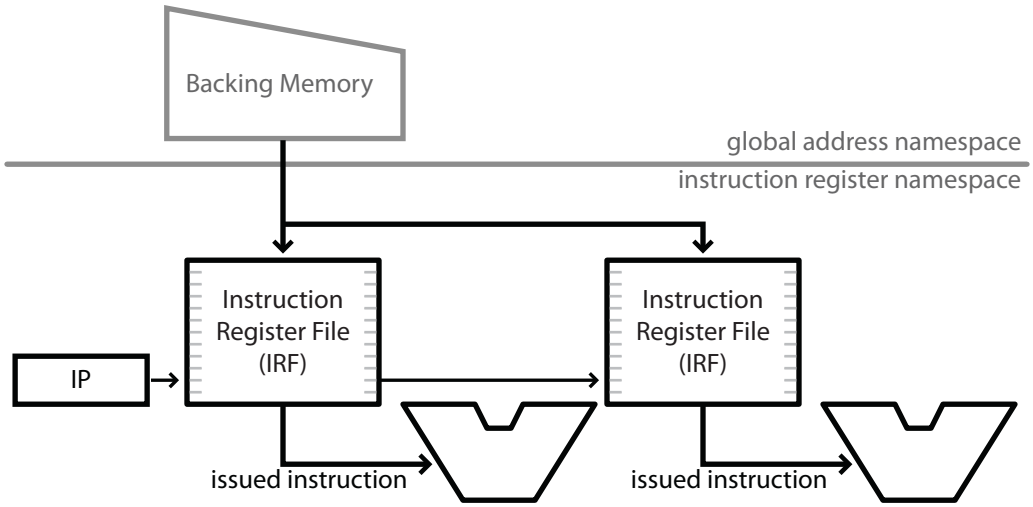


Figure 4.1 – Instruction Register Organization. Instruction registers are software-managed memory that let software capture critical instruction working sets close to the function units. The instruction registers form their own namespace, which differs from the address space used to identify instructions in the backing memory. Instruction registers are implemented as small, fast register files. The instruction register files are distributed to reduce the energy expended transferring instructions from the register files to the function units. Software orchestrates the movement of instructions from the backing memory to the instruction registers. The instruction pointer (IP) specifies the address of the next instruction to be issued from the instruction registers.

embedded applications comprise fewer than 1000 instructions and are smaller than 4 KB. Increasing the cache capacity reduces the number of instructions that must be fetched from the memories that back the instruction cache, which improves performance and often improves efficiency as well. It improves performance by reducing the number of stalls that occur when there is a miss in the instruction cache; it improves efficiency by reducing the amount of energy spent transferring instructions from backing memories, which are typically much more expensive to access. However, the energy expended accessing the first-level instruction cache increases with its capacity. In this way larger instruction caches reduce the average energy expended accessing instructions across large instruction working sets at the expense of increasing the energy expended accessing instructions within small working sets, such as a loop nest or kernel.

Elm reduces the cost of delivering instructions by extending the instruction storage hierarchy with a distributed collection of software-managed instruction register files (IRFs) to capture the small instruction working sets found in loops and kernels. The organization is illustrated in Figure 4.1. Essentially, we can reason about instruction registers as collections of very small software-managed caches residing at the bottom of the instruction memory hierarchy. We use the term register because, like their data register equivalents, instruction registers define a unique architectural namespace. Furthermore, instruction registers are explicitly managed by software, which is responsible for allocating instruction registers and scheduling load operations to transfer instructions into registers.

Unlike caches, instruction registers do not use tags to determine which instructions are resident. Issuing

an instruction stored in an instruction register does not require a tag check, and consequently instruction register files are faster and less inexpensive to access than caches of equivalent capacity. Instruction register files are much smaller than typical instruction caches. The instruction register files implemented in Elm provide 512 B of aggregate storage, enough to accommodate 64 instruction pairs, per processor. This allows the register files to be implemented close to function units, which reduces the cost of transferring instructions from instruction registers to data-path control points. The abundance of instruction reuse and locality within the loops and kernels that dominate embedded applications, whose important instruction working sets are readily captured in instruction registers, allows this organization to keep a significant fraction of instruction bandwidth local to each function unit. In exchange, the compiler must explicitly coordinate the movement of instructions into registers.

Executing Instructions Residing In Instruction Registers

Instructions are issued from instruction registers, and only those instructions that are stored in instruction registers may be executed. A distinguished architectural register named the instruction pointer (IP) designates the next instruction to be executed. It supplies the address at which instructions are read from the instruction register files (IRFs). The instruction pointer is similar to the program counter found in conventional architectures except that it only addresses the instruction registers and therefor is shorter than a conventional program counter. The instruction pointer is updated after an instruction issues to point to the next entry in the instruction register files. With the obvious exception of an instruction that explicitly sets the instruction pointer during a transfer of control, updating the instruction pointer simply involves incrementing it. Should the updated value lie beyond the end of the instruction register file, the value is wrapped around; this convention simplifies the construction of position independent code that will execute correctly regardless of where it is loaded in the instruction registers.

Unconditional jump instructions and conditional branch instructions allow control to transfer within the instruction registers. These instructions update the instruction pointer when they execute to reflect the transfer of control. Because an instruction must reside in an instruction register for it to be executed, the destination of a control flow instruction is always an instruction register. The destination may be specified using an absolute or relative position. Control flow instructions that use absolute destinations specify the name of the destination instruction register explicitly; instructions that use relative destinations specify a displacement that is added to the instruction pointer.

Returns and indirect jumps, the destination of which may not be known at compile time, use a register to specify their destinations. As with unconditional jump and conditional branch instructions, the destination instruction must reside in an instruction register. Unlike jump and branch instructions, the destinations of which are encoded in the instructions, the destinations of indirect jump instructions are read from registers when the instruction executes. Subroutine calls within the instruction registers use a jump and link instruction to store the return position in a distinguished register, the link register, when control transfers to the subroutine. The destination of the jump and link instruction must be an instruction that resides in an instruction register. An indirect jump to the destination stored in the link register transfers control back to the caller

when the subroutine completes.

Loading Instructions into Instruction Registers

Software explicitly coordinates the movement of instructions into instruction registers by executing instruction loads. An instruction load transfers a contiguous block of instructions from memory into a contiguous set of registers. Software specifies the size of the instruction block, where the block is to be loaded in the instruction register file, and where the block resides in memory. The size of the instruction block determines the number of instructions that are transferred when the load executes. The processor can continue to execute instructions while instructions load provided that control does not transfer to an instruction register with a load pending. Should control transfer to a register with a load pending, the processor stalls until the instruction arrives from the backing memory.

Instruction registers provide an effective mechanism for decoupling the fetching of instructions from memory and the issuing of instructions to the function units. This decoupling lets software hide memory access latencies. Software anticipates transfers of control, and proactively initiates instruction loads to fetch any instructions that are not present in the instruction registers.

Instruction loads may specify absolute or relative destinations. Loads that specify absolute destinations encode the name of the instruction register at which the block is to be loaded. Loads that specify relative destinations encode a displacement that is added to the instruction pointer to determine the destination. Allowing instruction loads to specify relative destinations simplifies the generation of position independent code that will execute correctly regardless of where it is loaded in the instruction registers.

Instruction loads may specify the memory address at which the instruction block reside directly or indirectly. Loads that specify the address directly encode the effective address of the instruction block in the instruction word. Loads that specify the address indirectly encode a register name and displacement in the instruction word; the effective address is determined when the load executes by adding the displacement to the contents of the specified register. The direct addressing mode is used when the location of the instruction block is known at compile time. This commonly occurs when an instruction block is transferred to a local software-managed memory before being loaded into instruction registers. The indirect addressing mode is used when the address of the instruction block cannot be determined at compile time. The indirect addressing mode is also used when the address of the instruction block is too large to be encoded directly. Additionally, the indirect addressing mode can be used to implement register indirect jumps by executing an instruction load followed by a jump. The instruction load uses the register that contains the address of the indirect jump target to load the target instruction; the jump transfers control to the target instruction after it has been loaded.

Contrasting Instruction Registers and Caches

Instead of instruction registers, we might consider using small instruction caches in the first level of the instruction memory hierarchy. These small caches are commonly referred to as filter caches. Like instruction registers, filter caches allow small instruction working sets to be captured close to the function units in memory that is less expensive to access than the first-level instruction caches typically used in most processors.

Perhaps the most obvious difference between instruction registers and filter caches is the use of tags to dynamically determine which instructions are present in a cache. More fundamentally, instruction registers and filter caches differ in the separation of management policy and responsibilities between software and hardware: instruction registers let software decide which instructions are displaced when instructions are loaded, which instructions are loaded together, and when instructions are loaded.

Naming

To request an instruction stored in a cache, a processor sends the complete address of the desired instruction to the cache. Some of the address bits are used to identify the set in which the block containing the desired instruction will reside if the instruction is present in the cache. Most of the address bits are used to check the block tag to determine whether the desired instruction is present, and most often the instruction is present in the cache. When the instruction is not present, the cache uses the address to determine the address of the cache block that needs to be requested from the backing memory. Because the entire address of the desired instruction is always needed to access the cache, the program counter must be wide enough to store a complete memory address.

To request an instruction stored in an instruction register, the processor sends a short instruction register identifier to the instruction register file. As described previously, the instruction pointer stores the identifier of the next instruction to be executed. Because there are relatively few instruction registers, the instruction pointer is short and less expensive to maintain than a program counter. The processor only needs to send a memory address to the instruction register file when loading instructions, an infrequent operation.

Placement and Replacement

The placement or mapping policy of a cache determines where a block may be placed within the cache. Consequently, it also affects, and may completely determine, which block is displaced when a block is loaded into the cache. Caches typically implement a fixed placement policy, though machines let software influence the placement policy. For example, some allow software to configure the set associativity of a cache [55]. When considering how the placement policy affects the cost of accessing an instruction cache, we need to consider the cost of accessing an instruction from the cache when the instruction is present in the cache, the cost of accessing an instruction when the instruction is not present in the cache and must be loaded from the backing memory, and the rate at which instructions are loaded from the backing memory. The cost of accessing an instruction that is not present in the cache is dominated by the cost of accessing the backing memory rather than storing the instruction in the cache.

Direct mapped caches allow a block to appear in only one place. Because only one entry is searched when a direct mapped cache is accessed, direct mapped caches usually store block tags in SRAM that is accessed in parallel with the instruction array. Set associative caches allow a block to appear in a restricted set of places. Accessing a set associative cache can consume more energy than accessing a direct mapped cache of the same capacity, as multiple ways may need to be searched, but the improvement in the cache hit rate can reduce the number of accesses to the backing memory enough to provide an overall reduction in the average access

energy. Set associative caches may store block tags in SRAM and access multiple tags in parallel. Highly associative caches with many parallel ways typically store the block tags in content addressable memories to allow all of the tags to be checked in parallel [105]. The match line can be used to drive the word-line of the instruction array so that the instruction array is only accessed when there is a hit. Fully associative caches allow a block to appear anywhere in the cache. Though impractical for large caches, filter caches are often small enough and require few enough tags to be fully associative.

Like fully associative caches, an instruction may be placed at any location in an instruction register file. Because software controls where an instruction load places instructions, software decides which instructions are replaced when new instructions are loaded. The elmcc compiler is often able to achieve effective hit rates that are competitive with or better than a cache by statically analyzing code to determine which instructions are best candidates for replacement. This is possible because a compiler can infer which instructions are likely to be executed in the future from the structure of code.

Similar optimization can be performed for a direct mapped cache when the compiler can determine when instructions will conflict in the cache. The code layout determines which instructions will conflict in the cache, and therefore the code layout phase must place instructions to avoid conflicts. Compilers may find it difficult to perform these types of code placement optimizations across function call boundaries. The address of a function provides both the name used by callers to identify the function and determines where it will reside in an instruction cache. To avoid conflicts between a function and its callers, the compiler must place the function such that it does not conflict with the locations at which it is called. This requires that a compiler consider both the function and its callers when placing instructions to avoid conflicts, which is similar in structure to register allocation. Instruction registers allow the caller to decide where the function will be loaded, and therefore allows the caller to decide which instructions are displaced. This greatly simplifies the problem of laying out code to avoiding conflicts, as it reduces it to a local problem of deciding where to load instructions in the instruction register rather than a global problem of deciding how to structure an application to reduce conflicts.

Because the number of instruction registers is expected to be small, it will often not be possible for a function and its caller to reside in the instruction registers without it being productive to inline the function. Instruction registers allow functions to be inlined without replicating the inlined function body at every call site. Instead, a single instance of the function can be loaded into the instruction registers at the inlined call locations. This avoids the problem of code expansion when function calls are inlined.

When a function is too large to be productively inlined and the compiler is able to bound the function's instruction register requirements, the compiler can allocate instruction registers to the called function and have the calling function restore the instruction registers when control returns. This convention is useful for leaf function calls. Because a leaf function does not call other functions, the compiler can easily bound its instruction register requirements.

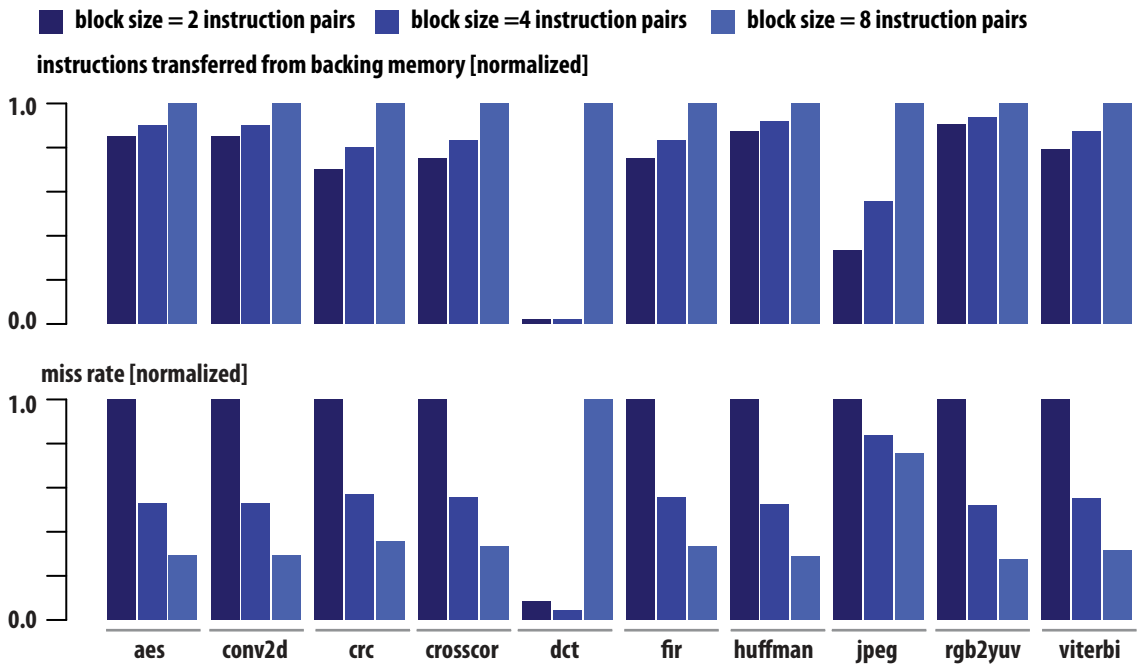


Figure 4.2 – Impact of Cache Block Size on Miss Rate and Instructions Transferred. The instructions transferred and miss rate are for a direct-mapped cache with a capacity of 64 instruction-pairs. The upper plot shows the relative number of instructions that are transferred from the backing memory; the lower plot shows the relative miss rate. The data in both plots are normalized within each kernel. The average number of instructions that are loaded from the backing memory increases with the block size, while the miss rate decreases. The abrupt increase in the `dct` kernel data results from conflict misses that occur when the block size is 8 instruction-pairs. The alignment of instructions within the kernel causes conflicts between instructions at the start and end of the loop. Consequently, there are 2 conflict misses per iteration of the loop despite the cache having enough capacity to accommodate the loop.

Block Size

The block size selected for a cache affects the number of tags that must be provisioned. It also affects the number of instructions that are loaded from the backing memory when there is a cache miss. Larger blocks better exploit any spatial instruction locality that is present in an application, and increasing the block size can reduce the miss rate when executing instruction sequences that exhibit significant spatial locality. The reduction in the miss rate improves performance, but does not reduce the number of instructions that are loaded from the backing store. Increasing the block size may increase the number of instructions that are loaded from the backing store. This occurs when the larger block size increases the number of instructions that are transferred to the cache but not executed before the block is replaced, as is illustrated by the data presented in Figure 4.2.

For a fixed cache capacity, increasing the block size reduces the number of tags, as each tag covers more instructions. Because the cost of searching the tags increases with the number of tags, increasing the block

size reduces the contribution of the tag search to the instruction access energy. However, increasing the block size can result in additional conflict misses. Because filter caches are small, and therefore hold relatively few blocks, the increase in the number of conflict misses caused when the block size is increased can be significant, as illustrated by the miss rate observed for the **dct** kernel in Figure 4.2.

Software-defined instruction blocks provide a flexible mechanism for specifying and encoding locality. The instructions in a block exhibit significant spatial and temporal locality, and typically exhibit significant reuse. Because software determines the size of the instruction block that is loaded by each instruction load operation, software can avoid inadvertently loading instructions that may not be executed when control enters the block. However, additional instructions must be executed to initiate the loading of instructions into instruction registers, which expands code sizes and partially neutralizes some of the reduction.

4.2 Microarchitecture

This section describes the instruction register architecture implemented in the Elm prototype and discusses design considerations that influenced the microarchitecture. Figure 4.3 illustrates in detail the microarchitecture of the instruction load (IL) and instruction fetch (IF) stages of the pipeline. Instructions are issued in pairs from the instruction register files, each of which has 64 registers. The instruction register files are two-port memories, with a dedicated read port and a dedicated write port. This allows the loading of instructions to overlap with the execution of instructions from the instruction registers.

The instruction register management unit (IRMU) coordinates the movement of instructions from the backing memory to the instruction registers. Instruction load commands are sent to IRMU from the XMU pipeline. The IRMU is decoupled from the function unit pipelines, which allows the processor to continue executing instructions while instructions are being loaded. The IRMU implemented Elm can handle a single instruction load at a time. Attempts to initiate a load operation while one is outstanding causes the pipeline to stall.

The instruction load unit implements the address generators and control logic that coordinates the transfer of instructions from the backing memory to the instruction registers. The instruction load unit has two address generators: one for generating memory addresses, and one for generating instruction register addresses. When an instruction block is loaded from a remote memory, and therefore must traverse the on-chip interconnection network, the memory address generator is only used to compute the address of the instruction block, and not the instructions within the block. The block address and the block size are sent to the local network interface, which performs a single block memory transfers to fetch the entire instruction block. This reduces the number of addresses that must be generated locally, and more importantly reduces the number of memory operations that traverse the network.

The IRMU uses a pending load (PL) scoreboard to track which instruction registers are waiting for instructions to arrive from the backing memory. When an instruction load is initiated, the IRMU updates the PL scoreboard to indicate that the destination registers are invalid. The scoreboard is updated when instructions arrive from the backing memory to indicate that the destination registers now contain the expected instructions. When the IRMU is loading instructions, the PL scoreboard is used to determine whether the requested

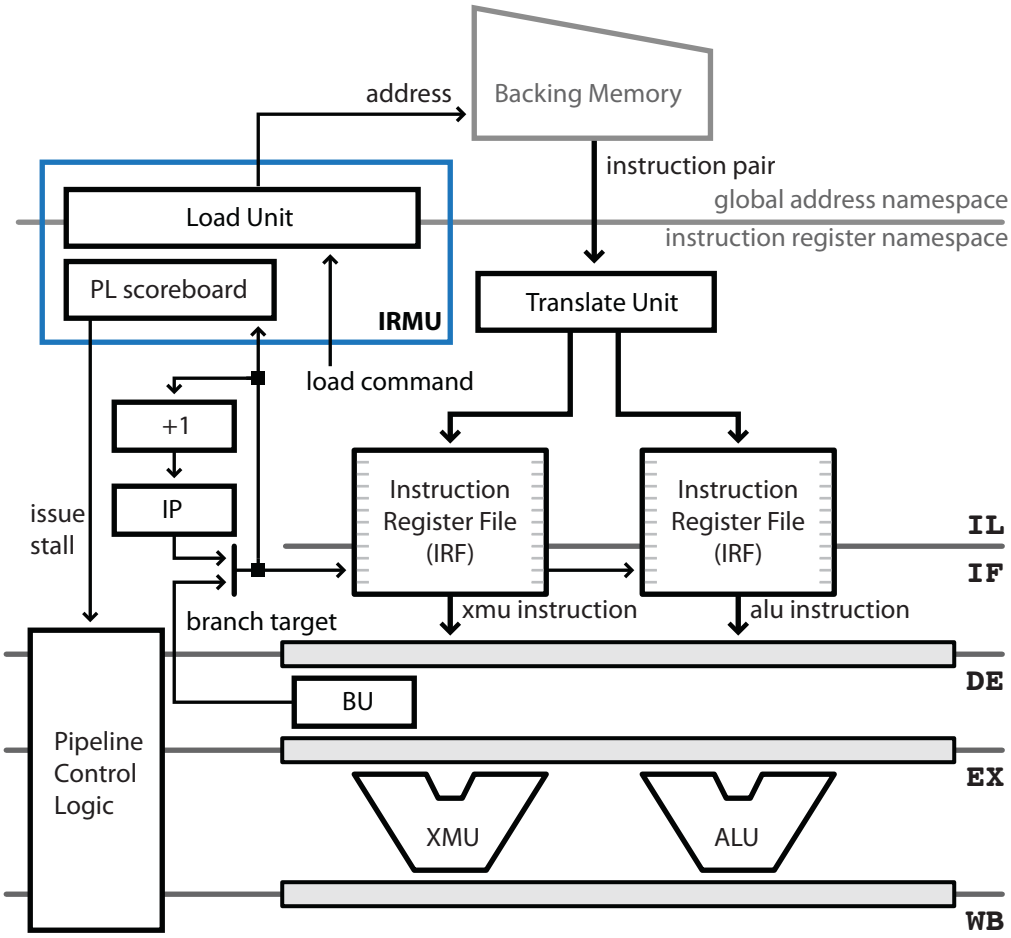


Figure 4.3 – **Instruction Register Microarchitecture.** The instruction register management unit (IRMU) coordinates the movement of instructions from the backing memory to the instruction registers. The IRMU uses a pending load (PL) scoreboard to track which instruction registers are waiting for instructions to arrive from the backing memory

instructions are available for issue. The same instruction pointer value is sent to the scoreboard and instruction registers, and the scoreboard and instruction registers are accessed in parallel. The scoreboard behaves like full-empty bits on the instruction register values, except that the scoreboard is only accessed when the IRMU is loading instructions so that energy is not expended accessing the scoreboard when the IRMU is idle.

The PL scoreboard in Elm is implemented as a vector of 64 registers, one per instruction register address. This allows the IRMU to track each instruction register pair independently, which is feasible because the number of instruction registers is small. Though the energy expended maintaining the scoreboard contributes to the cost of loading instructions, a more important design consideration is the impact of the scoreboard on the cycle time, particularly when designs use deeper instruction register files. The energy expended maintaining the scoreboard is a less significant concern because the scoreboard is only active when the IRMU

is loading instructions, which tends to be infrequent.

The size of the scoreboard can be reduced if it tracks contiguous sets of instruction registers rather than individual entries. Increasing the size of the sets that the scoreboard tracks will reduce its size, but the loss of resolution can cause the processor to stall unnecessarily during instruction loads. In the limit of a single block that covers all of the instruction registers, one bit could be used to track all of the registers. Such a simple scoreboard would be inexpensive to maintain and would offer fast access times, but the processor would be forced to stall during every instruction load until every instruction in the instruction block arrives from the backing memory. If such a scoreboard where used, the instruction register files could be implemented as single-port memories, as it would not be necessary to concurrently read one instruction register while writing a different register. Performance would suffer, but the instruction registers would be less expensive to access. Such a design would be appropriate when instruction reuse within the significant kernels of an application allows most instructions to be statically placed within the instruction registers.

Control flow instructions execute in the branch unit (BU), which is implemented in the decode stage and contains a dedicated link register. Conditional branch instructions are implemented as predicated jump instructions, with dedicated predicate registers used to control whether a branch is taken. When a jump and link instruction executes, the branch unit stores the return instruction pointer to the link register. The link register also provides the destination for a jump register instruction when it executes. The dedicated link register allows the branch unit to write the link register when a jump and link instruction is in the decode stage, before the instruction reaches the write-back stage, simplifying the bypass network. It also lets the branch unit read the operand to a jump register instruction before the instruction reaches the register files, reducing the delay associated with forwarding the destination to the instruction register files. When the desired destination of an indirect jump is not in the link register, the destination must be explicitly moved to the link register. In effect, the move instruction that updates the predicate register behaves like a prepare-to-branch instruction [48], explicitly informing the hardware of the intended branch destination. The value stored in the link register can be saved by moving it to a different register or storing it to memory.

The destinations of relative jump and branch instructions are resolved in the translation unit as they are loaded from the backing memory. Consequently, all control flow destinations are stored as absolute destinations within the instruction registers, which simplifies the branch unit because it does not need an address generator. Most processors resolve relative destinations when control flow instructions execute. Because an instruction is typically executed more than once after it is loaded, resolving the destination when instructions load reduces activity within the hardware that computes the destination address, thereby reducing the energy expended delivering instructions to the processor.

In addition to conditional branch control flow instructions, the branch unit also executes loop instructions. Loop instructions use predicate registers to control whether the loop transfers control. A subset of the predicate registers provide counters that can be used to control the execution of loops. The predicate field of an extended predicate register is set when the counter value is zero. When a loop instructions uses one of these extended predicate registers, the branch unit decrements the counter stored in the predicate register in the decode stage. Typically this reduces loop overheads by eliminating the bookkeeping instructions that update loop induction variables after each iteration of a loop. Updating the register that controls whether the branch

is taken early in the pipeline reduces the minimum number of instructions that must appear in a loop body to prevent dependencies on the predicate register from stalling the pipeline, which allows Elm to execute short loops efficiently. Software techniques for reducing the number of branch and overhead instructions such as loop unrolling, which replicates multiple copies of a loop body, provide similar benefits at the expense of expanded code sizes. Loop unrolling optimizations can be counterproductive with instruction registers because the increase in code size may prevent the instruction registers from accommodating important instruction working sets.

4.3 Examples

This section provides several examples that illustrate the use of instruction registers. To simplify the examples, the assembly listings that appear in this section show only those instructions that execute in the XMU function unit. To further simplify the examples, the delay slots and most of the instructions that do not affect the transfer of control and do not affect the management of the instruction registers are elided; those instructions that do not affect the transfer of control and instruction registers that are shown simplify serve as place markers in the code.

Loading and Executing Instructions

The assembly code fragment shown in Figure 4.4 provides a useful example for illustrating several basic concepts associated with the use of instruction registers to capture important instruction working sets. The assembly code as shown lacks instructions for loading instructions into instruction registers, which would be introduced during the instruction register allocation and scheduling compiler pass. The size of each basic block is annotated in the control-flow graph that appears in Figure 4.4. As illustrated by the control-flow graph, the code contains three basic blocks and one loop.

Before passing the code in Figure 4.4 to the assembler, the instruction register allocation phase of the compiler must allocate instruction registers and schedule instruction loads. Instruction register allocation and load scheduling are important phases in the compilation process. The allocation of instructions to registers affects efficiency by exposing or limiting opportunities for instruction reuse within different instruction working sets. The scheduling of instruction loads also affects performance. By anticipating transfers of control to instructions that are not in registers and initiating instruction loads before instructions are needed, the compiler can hide the latency associated with loading instructions into registers.

A simple though typically rather inefficient strategy for allocating and scheduling instruction registers is to load each basic block immediately before control transfers to it. This is readily accomplished by introducing jump-and-load instructions at the outgoing edges of each basic block in the control-flow graph. The resulting code can be improved using a simple optimization that removes load operations that are clearly redundant. An example of a trivially redundant load operations is one that load the instruction block in which it appears. In effect, this simple strategy implements an allocation policy that assigns all of the instruction registers to each basic block and a scheduling policy that defers instruction loads as late as possible.

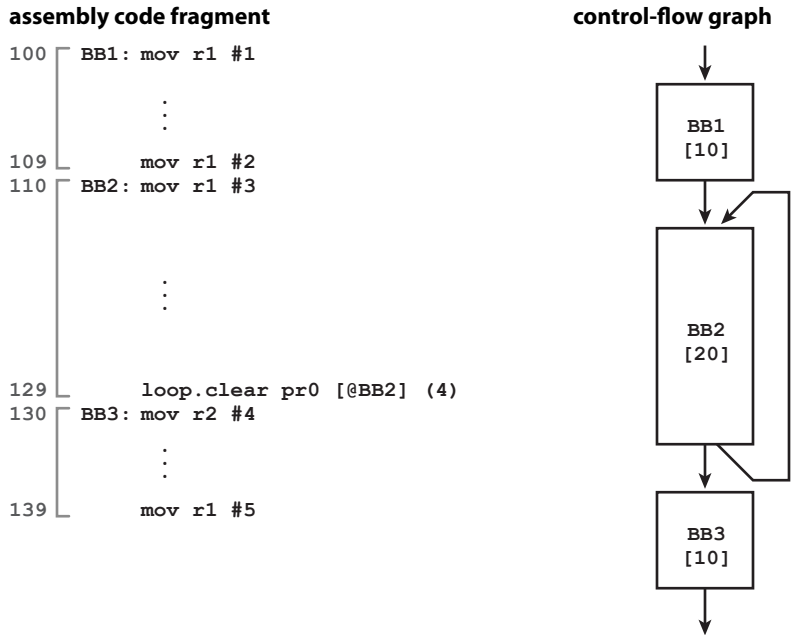


Figure 4.4 – **Assembly Code Fragment of a Loop that can be Captured in Instruction Registers.** The simplified assembly code contains a single loop of 20 instruction pairs, as the loop instruction at line 129 transfers control to the instruction pairs at line 110. Instructions that do not affect control flow have been replaced with unique `mov` instructions. Assembly statements at the boundaries of basic blocks are numbered in the code fragment, and the illustration of the corresponding control-flow graph shows the number of instructions in each basic block.

Figure 4.5 shows the assembly code fragment produced by the simple allocation and scheduling strategy after the redundant load elimination optimization is performed. Each jump-and-load instruction loads a new instruction block when control transfers between different basic blocks. The instruction blocks are illustrated at the right of the assembly code; the assembly statements that declare and define the instruction blocks are not shown to avoid cluttering the assembly code. For comparison, the basic block structure of the original code is illustrated at the left of the assembly code. There is a precise correspondence between the basic blocks in the original assembly code and the instruction blocks, as each instruction block encompasses a single basic block. The assembler allows assembly statements to use `@` expressions to specify the address of labels in the assembly code. The assembler also allows statements to use # expressions to refer to the length of an instruction block. For example, the jump-and-load instruction at line 100, which loads instruction block `IB1`, specifies that the instruction block resides at the memory address that the expression `@IB1` resolves to, and that the length of the instruction block is the size that `#IB1` resolves to.

Because the placement of the jump-and-load instructions in Figure 4.5 ensure that only one instruction block needs to be present in the instruction registers at any point in the execution of the code fragment, all of the jump-and-load instructions in Figure 4.5 use `ir0` as their targets. Consequently, each instruction load displaces instructions in the current basic block loaded instructions, and an instruction load operation may

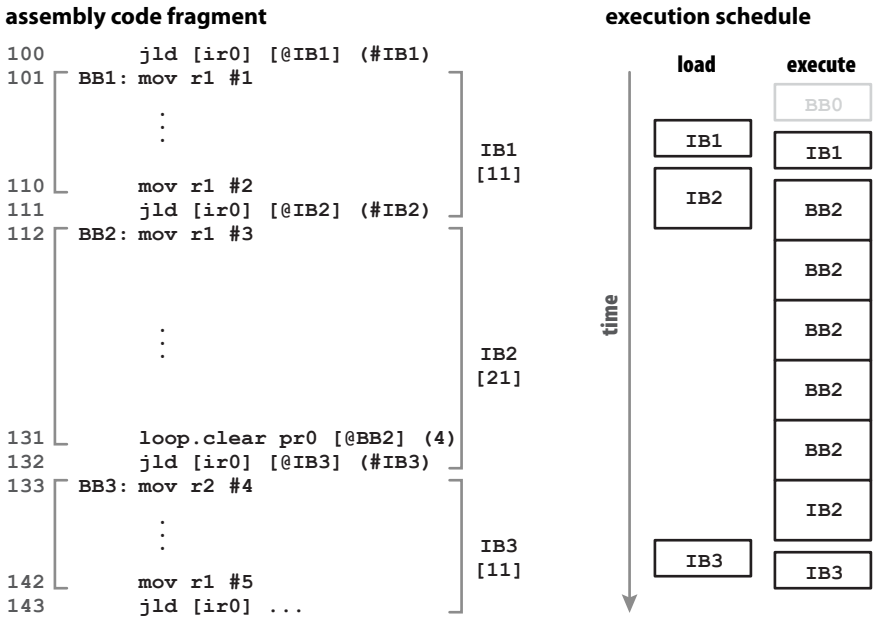


Figure 4.5 – Assembly Code Produced with a Simple Allocation and Scheduling Strategy. The corresponding instruction load and execution schedule is shown to the right of the assembly code.

displace the jump-and-load that initiated the operation. For example, the jump-and-load instruction at line 112 issues from instruction register ir10 and loads instruction block IB2 into ir0 through ir20.

The simple allocation and scheduling strategy is able to exploit what instruction reuse exists within the example. The loop involving basic block BB2 provides the only opportunity for instruction reuse, and the redundant load optimization ensures that the simple allocation and scheduling strategy loads the loop body only once.

However, the processor always stalls when a jump-and-load instruction executes because it must wait for the instruction register management unit to load the instruction that control transfers to. As illustrated in the execution schedule of Figure 4.5, using jump-and-load instructions to load each basic block individually and immediately before control transfers to it causes the processor to stall whenever control transfers to a different basic block. For example, execution stalls when control transfers from basic block BB1 to basic block BB2, and again when control transfers from BB2 to basic block BB3.

The compiler can eliminate some of the stalls in the execution schedule of Figure 4.5 by scheduling some instruction loads earlier, as illustrated in Figure 4.6. In the code shown in Figure 4.6, the jump-and-load instruction at line 100 loads basic blocks BB1, BB2, and BB3 as control transfers to basic block BB1. Figure 4.6 shows the resulting execution schedule. In addition to eliminating stalls, loading multiple basic blocks as a single instruction block reduces the number of instructions inserted by the compiler to manage the instruction registers. The code fragment in Figure 4.6 contains two fewer XMU instructions, which may improve the execution time and reduce the number of instructions that are transferred from the backing memory when the

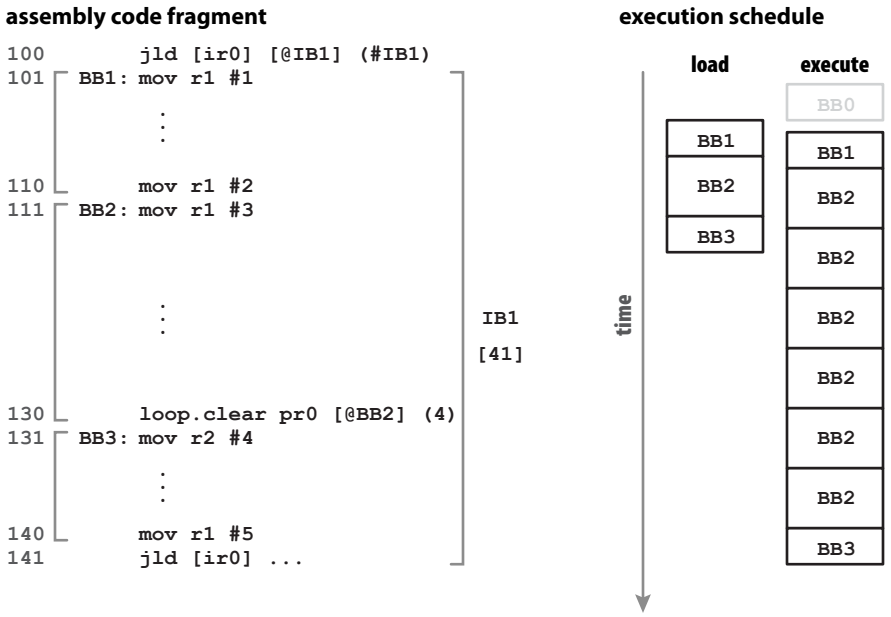


Figure 4.6 – **Assembly Code Fragment Produced when Instruction Load is Advanced.** The assembly code fragment shows the loop when the instruction load operations that load the individual basic blocks are scheduled before control enters the region. The load operations are merged into the single jump-and-load operation at line 100.

fragment is loaded into instruction registers.

Executing a Loop that Exceeds the Capacity of the Instruction Registers

The assembly code fragment shown in Figure 4.7 is similar in structure to the fragment used in the previous example. However, it differs in that the loop body exceeds the capacity of the instruction registers. The size of each basic block appears in the control-flow graph illustrated in Figure 4.7. As in the previous example, the code fragment has three basic blocks and one loop.

As before, the compiler must allocate instruction registers and schedule instruction loads before passing the code shown in Figure 4.7 to the assembler. Because the code fragment contains a basic block that exceeds the capacity of the instruction registers, the simple instruction register allocation and scheduling strategy used in the previous example cannot be applied.

Before proceeding to explain how the compiler should allocate and schedule instruction registers in this example, we should be aware that in some cases the compiler may be able to apply high-level code transformations such as loop fission to split large loops into multiple smaller loops, each of which will comprise few enough instructions to allow it to reside in instruction registers. After performing such high-level code transformations to restructure the loops within an application, the compiler may be able to allocate and schedule instruction registers within all of the loops in the application using the simple allocation and scheduling

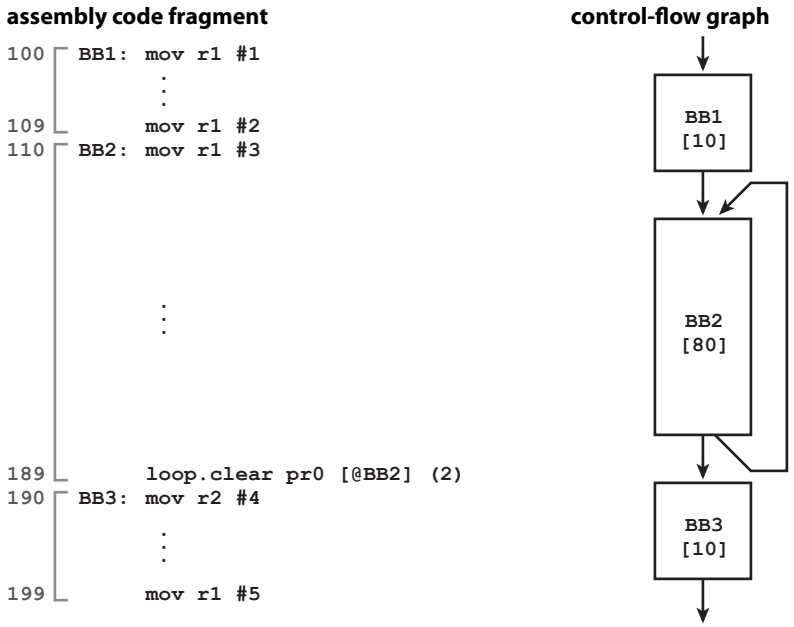


Figure 4.7 – **Assembly Code Fragment with a Loop that Exceeds the Instruction Register Capacity.** The assembly fragment contains a single loop with 80 instructions, exceeding the capacity of the 64 instruction registers. The illustration of the corresponding control-flow graph shows the number of instructions in each basic block.

strategy described in the previous example. The obvious and primary advantage of splitting larger loops is that the instructions comprising each loop would remain in instruction registers across all of the iterations of the loop, reducing the number of instructions that are transferred to instruction registers when the loops in the application execute. However, even when the compiler is able to split a loop, doing so may significantly increase the number of variables that need to be communicated between the portions of the loop. When the number of additional variables is large enough to require that some of the variables be stored in memory, the cost of storing the additional variables may offset any reductions in the instruction energy.

When the compiler cannot split a large loop or determines that it is not profitable to split a larger loop, it must load portions of the instructions comprising the loop as each iteration of the loop executes. A simple though inefficient strategy for allocating instruction registers and scheduling instruction loads is to partition the loop body into multiple smaller basic blocks and then load each basic block immediately before control enters it.

Figure 4.8 shows the assembly code after the compiler allocates instruction registers and schedules loads using this simple strategy. The instruction blocks are illustrated at the right of the code; the assembly statements that define the instruction blocks are not shown to avoid cluttering the code. Basic block `BB2`, which forms the body of the loop, is split into two smaller instruction blocks: `IB2a` and `IB2b`. The jump-and-load instruction at line `110` loads the first part of the loop before control enters the loop. The jump-and-load instruction at line `151` loads the second part of the loop before control enters it, displacing instructions in the

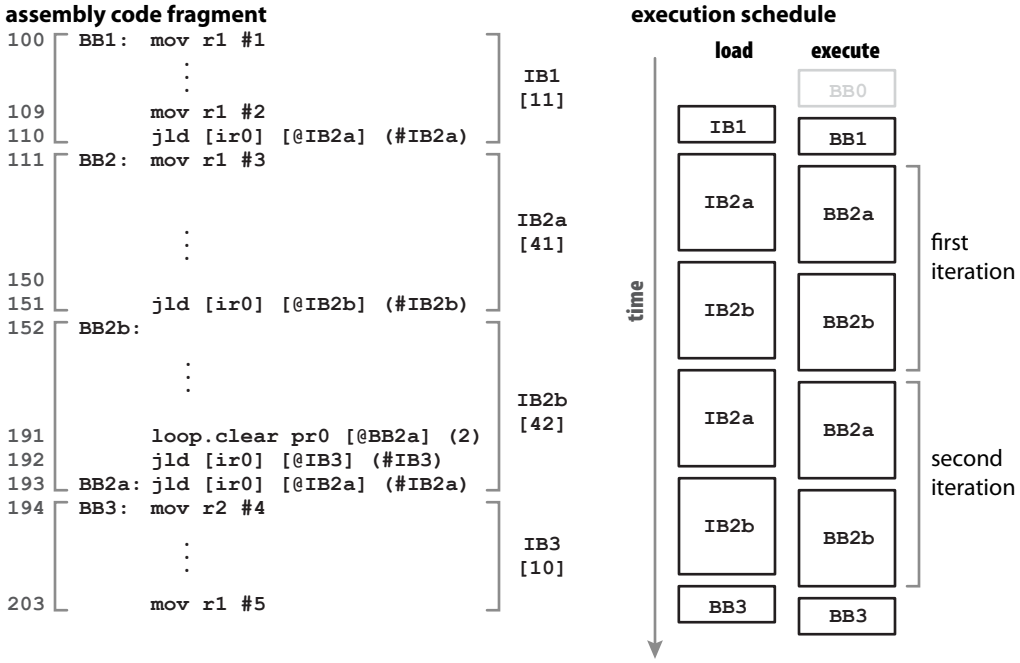


Figure 4.8 – **Loop after Basic Instruction Load Scheduling.** The assembly fragment shows the instruction blocks that result when instruction blocks are constructed by splitting large basic blocks and assigning each basic block to its own instruction block. The execution schedule illustrates the loading of instruction blocks and the execution of basic blocks when the assembly fragment executes.

first part of the loop. Because the first part of the loop is no longer present in the instruction registers when execution reaches the end of the loop, control cannot transfer directly to the start of the loop. Instead, the `loop` instruction at line 191 transfers control to the jump-and-load instruction at line 193, which loads the first part of the loop before transferring control.

The execution schedule is shown in Figure 4.9. As in the previous example, using jump-and-load instructions to load each basic block individually causes the processor to stall when control transfers to a different instruction block. In this example, the simple instruction register allocation and scheduling strategy also fails to capture instruction reuse across iterations of the loop, and every instruction in the loop is loaded during each iteration of the loop despite there being enough instruction registers to capture some instruction ruse across iterations.

The compiler can reduce the number of instructions that need to be loaded when executing loops that exceed the capacity of the instruction registers by allocating instruction registers so that some part of each loop remains in the registers across consecutive loop iterations. An effective strategy used by the compiler is to partition the body of the loop into multiple instruction blocks and assign one of the blocks an exclusive set of instruction registers. The other instruction blocks are assigned to the remaining instruction registers. The block that is assigned an exclusive set of instruction registers is loaded when control enters the loop. The remaining instruction blocks are loaded during each loop iteration. Essentially, this allocation and scheduling

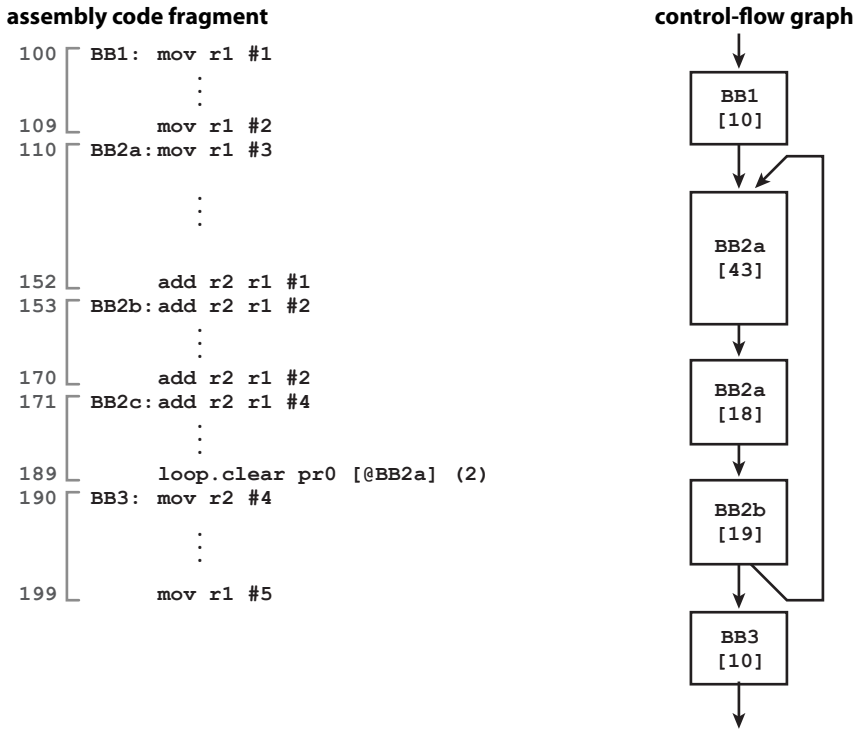


Figure 4.9 – **Assembly Code After Partitioning Loop Body Into 3 Basic Blocks.** The assembly fragment shows the basic blocks that result when the loop is partitioned into 3 basic blocks. The illustration of the control-flow graph shows the number of instruction pairs in each basic block.

strategy locks one of the instruction blocks in the instruction registers and sequences the remaining instructions through the remaining instruction registers.

Figure 4.9 shows the assembly code fragment with the body of the loop split into three basic blocks: `BB2a`, `BB2b`, and `BB2c`. The updated control-flow graph, annotated with the sizes of the basic blocks, is also illustrated in Figure 4.9. Though the sizes of the basic blocks might appear arbitrary, the partitioning of the loop body shown in the code fragment minimizes the number of instructions that are loaded during each iteration of the loop when the straight-line loop is split into three blocks. For now, we defer describing a more general approach to determining the number of blocks a loop should be split into and the size of the blocks until the subsequent section on instruction register scheduling and allocation, which follows the remaining examples.

Figure 4.10 shows the code produced by the compiler after instruction register allocation and scheduling. The instruction blocks are illustrated at the right of the assembly code; the assembly statements that define the instruction blocks are not shown to avoid cluttering the code. The jump-and-load instruction at line `110` loads basic block `BB2` and transfers control to the loop. The jump-and-load instruction at line `111` loads basic block `BB3` when the loop terminates. The code as shown assumes that the jump-and-load instruction at line `111` is loaded to instruction register `ir63`, which is preserved throughout the execution of the loop.

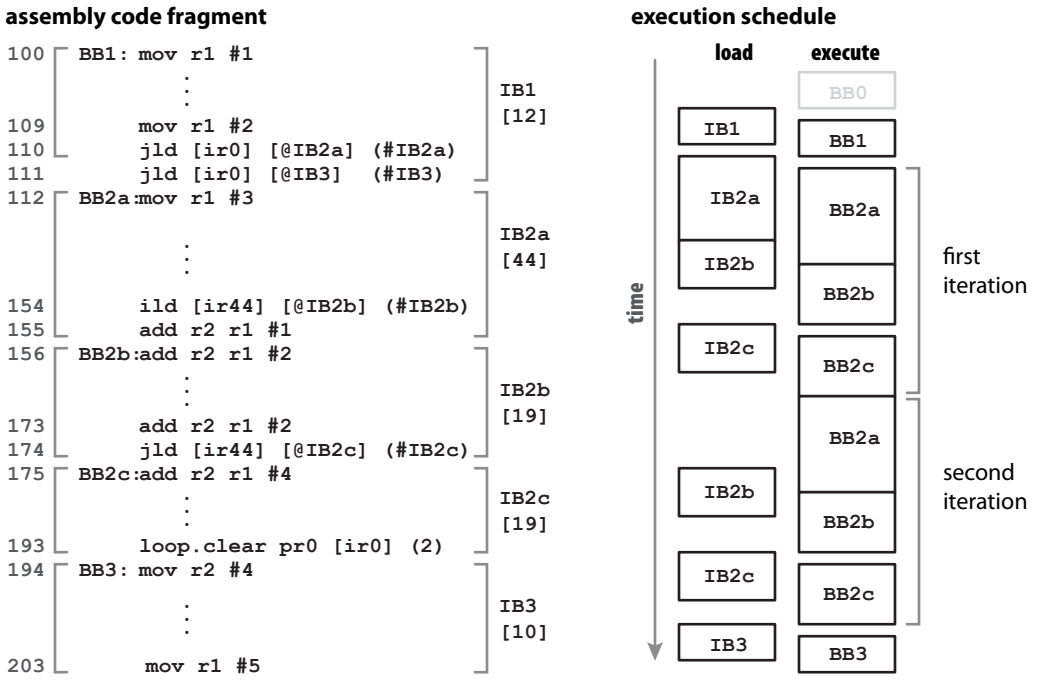


Figure 4.10 – **Assembly Code After Load Scheduling.** The assembly fragment shows the basic blocks and corresponding instruction blocks after instruction load and jump-and-load operations are scheduled. The execution schedule illustrates the loading of instruction blocks and execution of basic blocks when the assembly fragment executes.

Basic block BB2 is loaded when control first enters the loop and remains in the instruction registers until the loop completes. Basic blocks BB2a and BB2b are loaded during each iteration of the loop, and use the same instruction registers.

The ild instruction at line 154 loads basic block BB2b at instruction register ir44, immediately after BB2a in the instruction register file. This allows control to fall through to BB2b when the add instruction at line 155 executes from instruction register ir43. The load instruction at line 154 could appear earlier in the code to allow more time for the instruction register management unit to initiate the instruction load. The jump-and-load instruction at line 174 loads basic block BB2c and then transfers control to it. The loop instruction at line 193 executes from instruction register ir63. When the loop is repeated, the loop instruction transfers control to instruction register ir0, which contains the first instruction of BB2. When the loop terminates, control falls through to instruction register ir63, which contains the jump-and-load instruction at line 111. The jump-and-load instruction loads basic block BB3 and transfers control to it.

The execution schedule for the code is shown in Figure 4.10. As illustrated, instruction block IB2a is loaded once when control enters the loop and resides in the instruction registers for the duration of the loop. Instruction blocks IB2b and IB2c are loaded during every iteration of the loop. The code shown in Figure 4.10 loads 83 instructions during the first iteration of the loop and 38 instruction during each subsequent

source code fragment

```
100   int f(int x) {
101     return x + 1;
102   }
103
104   int g(int x) {
105     return f(x) + 1;
106   }
```

assembly code fragment

```
200  f: add r1 r2 #1                    ┐
201     jld [ir0] [r31] (1)             ┘ IBf1
202  g: mov r30 r31                     ┐
203     mov r31 @IBk2                   │ IBg1
204     jld [ir0] [@IBf] (#IBf1)        ┘
205     jld [ir0] [@IB3] (#IBg3)        ┐ IBg2
206     add r1 r1 #1                    ┐
207     mov r31 r30                     │ IBg3
208     jld [ir0] [r31] (1)             ┘
```

Figure 4.11 – **Assembly Code Fragment Illustrating a Procedure Call.** The instruction at line 203 stores the return address before the jump-and-load instruction at line 204 transfers control to `f`. The jump-and-load instruction at line 201 transfers control to g when `f` returns.

iteration. Note that the 83 instructions loaded in the first iteration include the jump-and-load instruction at line 111, which is the last instruction in the loop. In contrast, the code shown in Figure 4.9 loads 83 instruction during every iteration of the loop.

As illustrated in Figure 4.10, the register allocation and scheduling strategy used in Figure 4.10 reduces the number of stalls. The instruction load at line 154 initiates the load of instruction block `IB2b` before control reaches basic block `BB2b`, thereby allowing control to fall into basic block `BB2b` without requiring that the processor stall while it waits for the instruction register management unit to load the next instruction to be executed. Similarly, because basic block `BB2a` remains in the instruction registers across loop iterations, the loop instruction at line 193 can transfer control to basic block `BB2a` without the processor stalling. The only stall encountered during the execution of the loop body is the stall imposed by the jump-and-load instruction at line 174 that loads `IBB2c` as control enters basic block `BB2c`.

Implementing Indirect Jumps and Procedure Calls as Stubs

When performing an indirect jump, the number of instructions that should be loaded may not be known at compile time. For example, an indirect jump that implements a switch statement may jump to case statements of different length. Similarly, an indirect jump that implements a virtual function call or a function call that uses a function pointer to determine the address of the function may jump to functions of different length. When the number of instructions that will need to be loaded cannot be determined precisely at compile time, the compiler may load a subset of the instructions before executing the jump and defer loading any additional instructions until after transferring control. For example, switch statements can be implemented using indirect instruction loads to load the initial instructions of the target case statement, deferring the loading of additional instructions until after control transfers to the case statement. To reduce the number of loads that execute, the compiler can set the size of the initial load to be equal to the shortest case statement so that only longer case statements perform additional loads. Similarly, indirect jumps that implement virtual function calls can initially load the function prologue and defer loading the remainder of the function body until after control transfers to the callee function.

Figure 4.11 illustrates an assembly code fragment that implements a function call and return using a

source code fragment

```
100   int f(int x) {
101     return x + 1;
102   }
103
104   int g(int x) {
105     return x + 2;
106   }
107
108   int h(int (*p)(int)) {
109     return p(1);
110   }
```

assembly code fragment

```
200  f: add r1 r2 #1                    ┐ IBf1
201     jld [ir0] [r31] (1)              ┘
202  g: add r1 r2 #2                    ┐ IBg1
203     jld [ir0] [r31] (1)              ┘
204  h: mov r1 r2                       ┐
205     mov r2 #1                        │ IBh1
206     jld [ir0] [r1]  (2)              ┘
```

Figure 4.12 – Assembly Code Fragment Illustrating Function Call Using Function Pointer. Function h applies the function passed as p to the constant value 1. Because the function call at line 109 is a tail call, the return address passed to h in register r31 is passed unmodified to the function called through p.

simple calling convention in which the caller specifies where control should return to when the invoked function completes. The assembly code fragment assumes that both f and g may be called from multiple places in the application, and therefore the return addresses cannot be determined during compilation.

The calling convention illustrated in Figure 4.11 implements function calls as a jump-and-load instruction that loads the function entry and then transfers control to it. The calling convention implements returns as a jump-and-load instruction that loads one instruction at the return address to instruction register ir0 and then transfers control to it. The instruction at the return address then loads additional instructions. The return address is passed in register r31.

Function f receives its argument in register r2 and returns its result in register r1. The return address is passed in register r31, and the jump-and-load instruction at line 201 implements the return. It loads a single instruction from the return address passed in r31 and transfers control to it.

The implementation of g is somewhat more complex because it calls another function. As before, g expects the caller to pass a return address in register r31. The first instruction in g at line 202 saves the return address. The second instruction at line 203 moves the address to which the call of f should return into r31. The jump-and-load instruction at line 204 transfers control to f. Because the call target can be resolved statically when the compiler processes the code fragment shown in Figure 4.11, the jump-and-load instruction that implements the call of f can load f in its entirety. The jump-and-load instruction at line 205 is the return target that is passed to f. It loads the remaining instructions of g. The instructions at lines 207 and 208 restore the return address passed to g and then return to the calling context using a jump-and-load instruction.

Figure 4.12 illustrates a function call that uses a function pointer to determine the address of the called function. The example assumes that the compiler is able to determine that the argument to h is either f or g. The structure of the assembly code generated for functions f and g is identical to that generated for the function f shown in Figure 4.11. The assembly statement at line 204 moves the address of the function pointer passed as the argument to h to register r1. The assembly statement at line 205 moves the argument to the function call at line 109 to register r2, which is the register that f and g receive their argument from. The jump-and-load at line 206 loads the function passed as the argument to h, which is either f or g, and

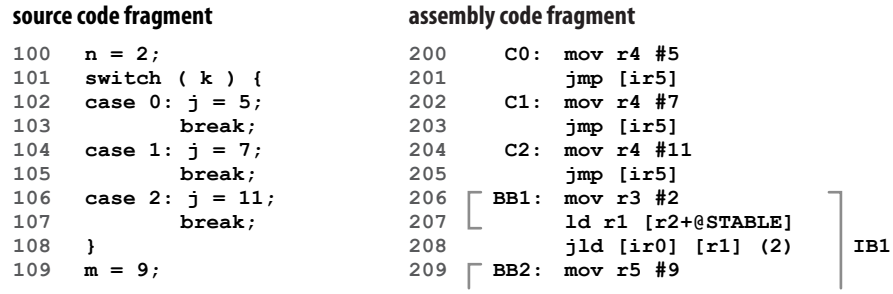


Figure 4.13 – Assembly Code Illustrating Indirect Jump. The assembly code fragment implements the switch statement at line 101 using an indirect jump-and-load. Instruction block IB1 is loaded at instruction register ir0, and the instruction at 209 has been scheduled to execute in the delay slot of the jump instruction at 208. The load operation at line 207 loads the address of the target case statement block using the value of k in register r2. The addresses of the case statement blocks are stored in the @STABLE vector. The break statements at lines 103, 105, and 107 are implemented as jump operations at lines 201, 203, and 205.

then transfers control to the called function. The size of f and g are known at compile time and are the same, which allows the jump-and-load at line 206 that implements the function call to load the entire body of the called function. Because h immediately returns after the function called through p completes, the return address passed to b in register r31 is left unmodified. Consequently, the jump-and-load statements at lines 201 and 203 that implement the return statements in f and g transfer control to the return address passed to h.

Figure 4.13 illustrates the use of jump-and-load instructions to implement a switch statement. The assembly generated for the case clauses of the switch statement appear at lines 200 through 205. The assembly statements implementing the switch statement appear at lines 207 and 208. The argument to the switch statement, variable k, resides in r2. The addresses of the case clauses are stored in a table at address STABLE that is indexed by k to determine the address of the case clause to be executed. The address is loaded into register r1 at line 207, and then the jump-and-load statement at line 208 loads the instructions comprising the case clause into the instruction registers at ir0 and transfers control. The assembly code shown assumes that instruction block IB1 is loaded at instruction register ir2, so the case clauses transfer control to the statement after the switch statement by jumping to instruction register ir5, which holds the instruction produced by the assembly statement at line 209.

Remembering Which Instructions Have Been Loaded

Figure 4.14 shows an assembly code fragment that contains a loop with an if – else statement in the body of the loop. The control-flow graph illustrated in Figure 4.14 shows the size of each basic block before the compiler performs instruction register allocation and scheduling. As is illustrated, the loop initially contains 70 instructions and exceeds the capacity of the instruction registers. Consequently, some of the instruction registers must be assigned to multiple instructions within the loop body. The instruction register allocation used to generate the assembly code assumes that basic block BB2 is more likely to be executed than BB3 and

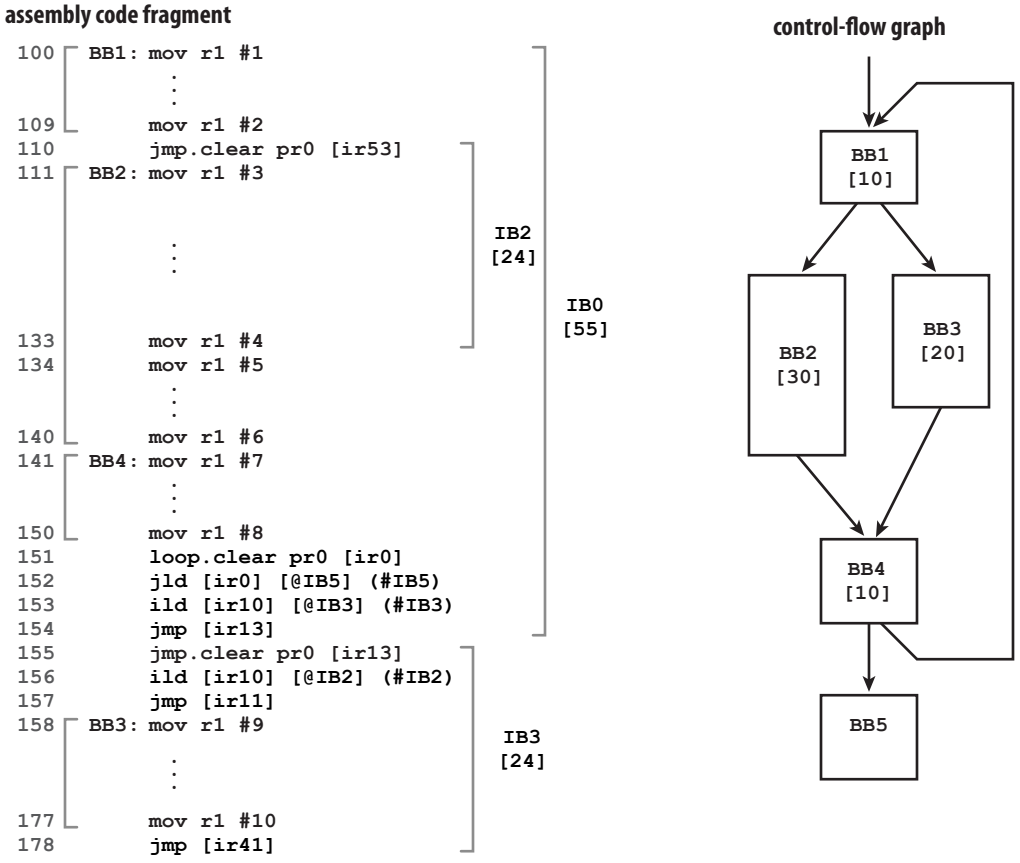


Figure 4.14 – **Assembly Fragment Containing a Loop with a Control-Flow Statement.** The loop comprises basic blocks BB1, BB2, BB3, and BB4, which exceed the capacity of the instruction registers. The illustration of the control-flow graph shows the number of instructions in each basic block.

that consecutive iterations of the loop are likely to follow the same path through the loop. The instructions introduced by the compiler during instruction register allocation and scheduling allow software to remember whether BB2 or BB3 resides in the instruction registers so that instructions are not loaded when already present in the registers. After instruction register allocation and scheduling, the loop contains 79 instructions.

Figure 4.15 shows the contents of the instruction registers when BB2 is present and when BB3 is present. Basic blocks BB1 and BB4 are present in both cases and remain in the instruction registers across all iterations of the loop. The conditional jump statement in register ir10 determines whether control falls through to BB2 or conditionally transfers to BB3. The destination of the conditional jump indicates whether BB2 or basic block BB3 is present in the instruction registers, and is updated each time a different basic block is loaded.

Basic block BB2 is loaded when control initially enters the loop. When basic block BB2 is present in the instruction registers, it resides immediately after the conditional jump statement. When the conditional jump in register ir10 is not taken, control falls through to basic block BB2. After basic block BB2 completes, control falls into basic block BB4, where the loop statement in register ir51 transfers control back to ir0

Instruction Registers with BB2 Loaded

```
00 ┌ BB1: mov r1 #1
   │        :
   │        :
09 └      mov r1 #2
10        jmp.clear pr0 [ir53]
11 ┌ BB2: mov r1 #3
   │        :
   │        :
   │
   │
33 │      mov r1 #4
34 │      mov r1 #5
   │        :
   │        :
40 └      mov r1 #6
41 ┌ BB4: mov r1 #7
   │        :
   │        :
50 └      mov r1 #8
51        loop.clear pr0 [ir0]
52        jld [ir0] [@IB5] (#IB5)
53        ild [ir10] [@IB3] (#IB3)
54        jmp [ir13]
```

Instruction Registers with BB3 Loaded

```
00 ┌ BB1: mov r1 #1
   │        :
   │        :
09 └      mov r1 #2
10        jmp.clear pr0 [ir13]
11        ild [ir10] [@IB2] (#IB2)
12        jmp [ir11]
13 ┌ BB3: mov r1 #9
   │        :
   │        :
   │
32 └      mov r1 #10
33        jmp [ir41]
34        mov r1 #5
   │        :
   │        :
40        mov r1 #6
41 ┌ BB4: mov r1 #7
   │        :
   │        :
50 └      mov r1 #8
51        loop.clear pr0 [ir0]
52        jld [ir0] [@IB5] (#IB5)
53        ild [ir10] [@IB3] (#IB3)
54        jmp [ir13]
```

Figure 4.15 – **Contents of Instruction Registers During Loop Execution.** The left half of the figure shows the contents of the instruction registers with basic block BB2 loaded.The right half of the figure shows the contents of the instruction registers with basic block BB3 loaded.

while subsequent iterations of the loop remain to be executed. When the conditional jump in register ir10 is taken, control transfers to the instruction residing in instruction register ir53. The instruction load instruction residing in ir53 loads instruction block IB3, which contains basic block BB3, at instruction register ir10. The subsequent jump statement transfers control to basic block BB3. Note that instruction block IB3 modifies the conditional jump instruction residing in instruction register ir10 so that control transfers to instruction register ir13 rather than instruction register ir53 when the conditional jump transfers control.

When basic block BB3 is present in the instruction registers, it occupies a subset of the instruction registers assigned to basic block BB2. The jump instruction in instruction register ir33 transfers control to basic block BB4 after basic block BB3 completes. The instructions loaded into instruction register ir11 and ir12 along with BB3 when instruction block IB3 is loaded displace the initial 2 instructions of basic block BB2. These instructions replace the instruction from BB2 that are displaced by instructions in basic block BB3 and restore the conditional jump instruction in instruction register ir10.

4.4 Allocation and Scheduling

This section describes several strategies for allocating instruction registers and scheduling instruction loads. The initial strategy is simple, requires minimal sophistication within the compiler, and is readily explained. Subsequent strategies become progressively more sophisticated and are better able to capture instruction

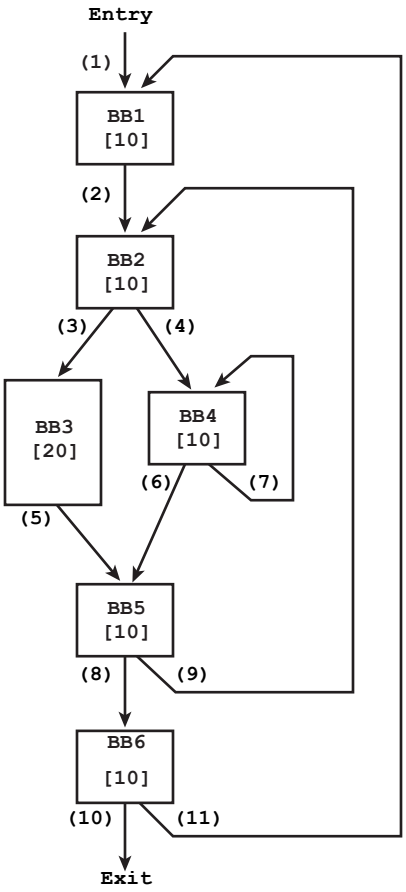


Figure 4.16 – **Control-Flow Graph.** The control-flow graph shows the structure of the kernel.

reuse within the instruction registers. Figure 4.16 provides a control-flow graph that will be used to illustrate the instruction register allocation and load scheduling strategies described in this section. The control-flow graph corresponds to some kernel that would typically be invoked repeatedly in place, with each invocation processing new input, so we might imagine an edge connecting the `Exit` node of the graph to the `Entry` node.

A Simple Allocation and Scheduling Strategy

Perhaps the simplest strategy for allocating instruction registers and scheduling instruction loads is to assign instruction registers to each basic block independently and defer loading the block until immediately before control transfers to it. Because this allocation and scheduling strategy makes all of the instruction registers available to each basic block, each basic block within a kernel or function can be loaded at instruction register `ir0`, and the compiler can use a jump-and-load instruction both to load the basic block and to transfer control.

The compiler can implement the allocation and scheduling strategy as three phases. The first phase prepares the control-flow graph for allocation and scheduling. The compiler breaks large basic blocks that exceed the capacity of the instruction registers into multiple smaller blocks and determines the relative positions of the basic blocks in memory.

The second phase computes an initial load schedule. The compiler tentatively schedules jump-and-load operations at each edge in the control-flow graph. The jump-and-load operation scheduled at the edge from basic block BBx to BBy loads the successor basic block BBy at instruction register ir0. For example, the compiler would schedule jump-and-load operations to load basic block BB2 at edges (2) and (9), a jump-and-load operation to load BB3 at (3), and a jump-and-load operation to load BB5 at (8).

The third phase improves the initial load schedule using a pair of optimization passes. The first optimization pass eliminates redundant loads. A jump-and-load operation is redundant if the instruction block it loads is already present at the desired location. For example, the jump-and-load operation inserted at (7) is redundant because basic block BB4 already resides at ir0 when control reaches (7). The second optimization pass merges load operations. Two jump-and-load operations can be merged if the later operation is dominated by the earlier and the corresponding instruction blocks are contiguous in memory. For example, the jump-and-load operation inserted at (5) in the example control-flow graph shown in Figure 4.16 can be merged into the jump-and-load operation inserted at (3) if basic blocks BB3 and BB5 are contiguous in memory. This optimization reduces the code size and the number of jump-and-load instructions that are executed.

This simple allocation and scheduling strategy fails to exploit much of the instruction reuse that exists within embedded kernels, though it succeeds in capturing a reasonable amount of the spatial locality. Instruction reuse is limited to loops composed of a single basic block, such as BB4 in the example control-flow graph. The optimization pass used to merge instruction loads allows the compiler to exploit spatial locality across adjacent basic blocks. To exploit instruction reuse within larger instruction working sets, the compiler must allocate and schedule instruction registers across multiple basic blocks, as described in the next section.

An Improved Region-Based Allocation and Scheduling Strategy

The simple allocation and scheduling strategy described above fails to capture short-term instruction within loops composed of multiple basic blocks because it assigns conflicting sets of instruction registers to the basic blocks within a loop. This deficiency is readily addressed by assigning basic blocks within some meaningful portion of the code to different sets of instruction registers. For example, the compiler can allocate disjoint sets of instruction registers to the instructions comprising the loop composed of basic blocks BB2 – BB4 by visiting the basic blocks in a depth-first traversal order and assigning the set of instruction registers to a basic block when it is visited. If the depth-first traversal visits BB3 before BB4, the allocator assigns instruction registers ir0 – ir9 to BB2, instruction registers ir10 – ir29 to BB3, instruction registers ir30 – ir39 to BB5, and instruction registers ir40 – ir49 to BB4. This assignment allows the entire loop body to remain in the instruction registers for the duration of the loop, significantly improving the amount of instruction reuse captured in the instruction registers. A region-based allocation and scheduling strategy allows the compiler to capture more instruction reuse within the instruction registers than the simple allocation and scheduling

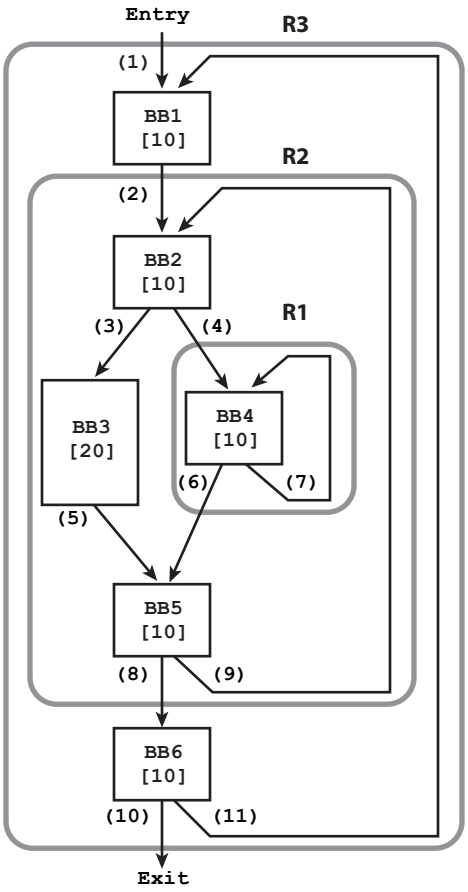


Figure 4.17 – **Illustration of Regions in the Control-Flow Graph.** In addition to 3 regions that are explicitly illustrated and labeled, each basic block comprises a region.

strategy described above.

Informally, a region of a flow graph is a portion of the flow graph with a single point of entry. The single point of entry provides a convenient location for the compiler to load the instructions in the region. More formally, a region is a set of nodes N and edges E in a flow graph such that there is a single node d that dominates all of the nodes in N, and if node n_1 can reach n_2 in N without passing through d then n_1 is also in N. A procedure will have a hierarchy of regions, with the basic blocks in the procedure forming leaf regions. Using the hierarchy of regions in a procedure to reason about the allocation and scheduling of instruction registers within the compiler is productive because each statement in a block-structured procedure corresponds to a region, each level of statement nesting in the procedure corresponds to a level in the hierarchy of regions, and each of the natural loops in a procedure establishes a unique region. Figure 4.17 illustrates the regions corresponding to the natural loops in Figure 4.16.

The region-based allocation and scheduling strategy first processes regions defined by the natural loops in the procedure. Those regions that are not part of a natural loop are processed last because there is limited

or no opportunity for instruction reuse within these regions. The allocator sorts the regions so that smaller regions that are closer to the leaf regions in the hierarchy of regions are processed before larger enclosing regions. This ensures that instructions in inner loops are assigned instruction registers before instructions in enclosing loops.

Allocating instruction registers within larger regions allows the instructions in the regions to be assigned to contiguous sets of instruction registers. This reduces the number of instruction load operations that are required to load composite instructions regions. Before processing a region, the allocator attempts to expand the allocation region into the immediately enclosing region. A region is expanded when the enclosing region is small enough to fit in the instruction registers. For example, when processing the procedure illustrated in Figure 4.17, the allocator would begin with process region `R1`. The allocator would expand the allocation region to `R2`, which contains 50 instructions. The allocator cannot further expand the allocation region because region `R3`, the enclosing region in the hierarchy of regions, contains 70 instructions and exceeds the capacity of the instruction registers.

After expanding the allocation region, the allocator tentatively assigns instruction registers to the basic blocks in the region based on a depth-first traversal of the basic blocks in the region. The assignment is tentative in that it may later be repositioned, though the relative positions of the instructions within the region will not change. For example, the allocator would tentatively assign instruction registers `ir0` – `ir49` to the basic blocks in region `R2`. Instruction registers `ir0` – `ir9` would be assigned to `BB2`, instruction registers `ir10` – `ir29` to `BB3`, instruction registers `ir30` – `ir39` to `BB4`, and instruction registers `ir40` – `ir49` to `BB5`.

When a region exceeds the capacity of the instruction registers, the allocator attempts to assign instruction registers not used in any of the enclosed regions that were previously processed and assigned registers. For example, region `R3` contains more instructions than there are instruction registers. When processing region `R3`, the allocator assigns instruction registers `ir0` – `ir9` to basic block `BB1`, translates the assignments within region `R2` so that the region uses registers `ir10` – `ir59`, and assigns instruction registers `ir60` – `ir63` and `ir0` – `ir5` to basic block `BB6`.

After the instruction register allocation phase completes, the load scheduling phase inserts load operations. The scheduler first tentatively schedules jump-and-load operations at each basic block, as in the previous allocation and scheduling strategy. The scheduler then schedules load operations to load regions that were allocated contiguous sets of instruction registers at the edges arriving at the dominator node. These load operations are inserted to render some of the tentatively scheduled jump-and-load operations redundant. Generally, this strategy of lifting loads closer to the entry point of the procedure is productive because it reduces the number of instruction load operations. However, this strategy may result in some regions being loaded and not executed, such as when there is a point of divergence in the control-flow graph. The scheduler can defer loading regions that are not certain to be executed until control transfers to the region to avoid this.

Redundant load operations are identified using a data-flow analysis that determines which instructions are certain to reside in the instruction registers at each point in the code. The values computed by the data-flow analysis are sets of instructions, referred to as the resident instruction sets. The data-flow analysis initializes the resident instruction sets to the empty set. The transfer function used in the data-flow analysis updates the resident instruction sets when load operations are encountered. The instructions that are loaded by the

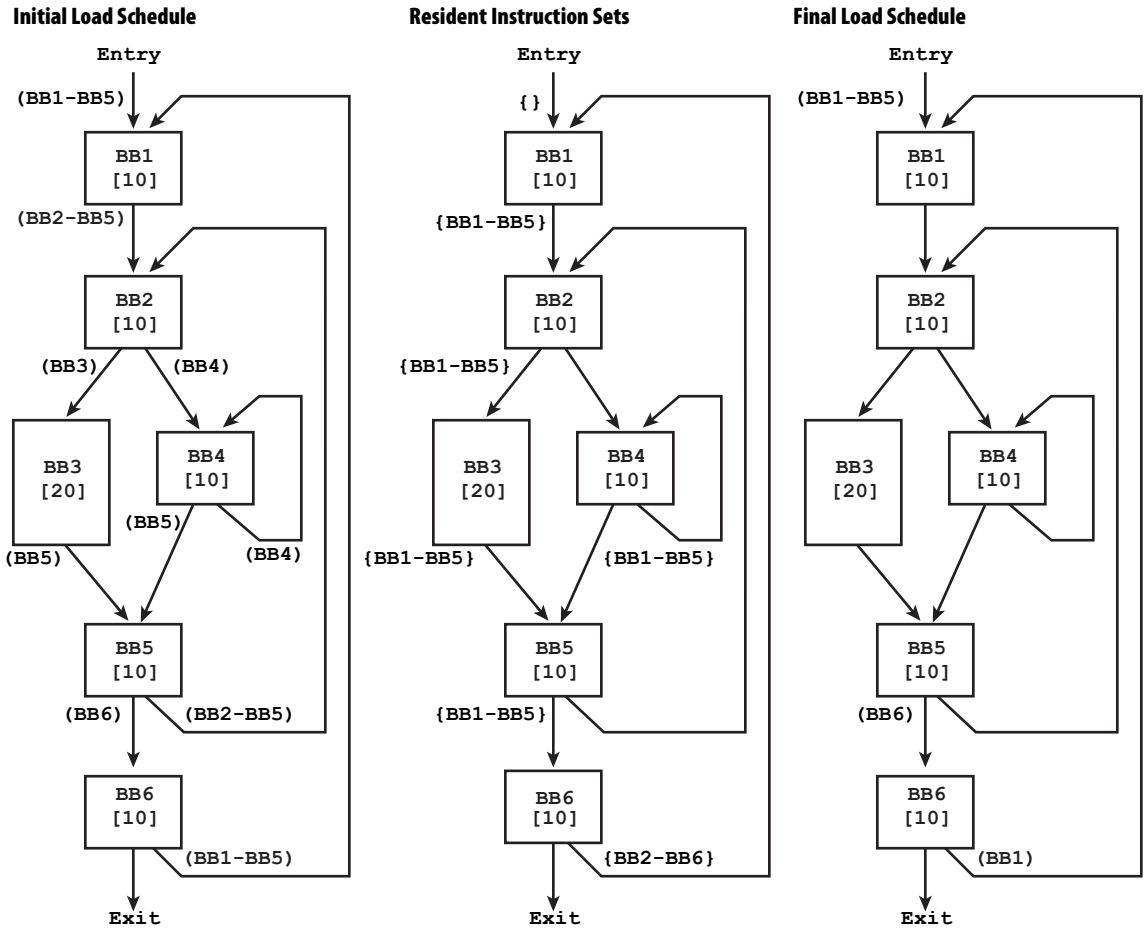


Figure 4.18 – **Resident Instruction Sets.** The control-flow graph at the left shows the initial load schedule. The edges of the control-flow graph are annotated with the basic blocks that the compiler has scheduled to be loaded when control transfers between the basic blocks incident on the edge. The resident instruction sets for the initial load schedule are shown on the control-flow graph in the middle. Each basic block is annotated with the set of basic blocks that are in the resident instruction set computed for the basic block. The control-flow graph at the right shows the revised load schedule.

load operation are added to the resident instruction set, and any instructions that are displaced by the load operation are removed. The meet operator used to combine data-flow values at convergence points computes the intersection of the resident instruction sets over the converging paths. This ensures that only those instructions that are present at all incoming paths are added to the resident instruction set. After computing the resident instruction sets, the load scheduler optimizes the load schedule by eliminating any loads that are redundant. A load is redundant if it loads an instruction block that is already present, as computed by the resident instruction set data-flow analysis. Figure 4.18 illustrates the initial load schedule, the resident instruction sets computed by the data-flow analysis, and the optimized load schedule.

Improving Instruction Reuse Within Large Allocation Regions

The amount of instruction reuse captured in the instruction registers when executing loops that exceed the capacity of the instruction registers can be improved by extending the region-based allocation strategy described above. The basic idea is to partition the instruction registers into two disjoint sets. Registers in the first set are assigned to instructions that reside in registers for the duration of the loop. Registers in the second set are assigned to instructions that are loaded while the loop executes. We sometimes describe registers in the second set as *non-exclusive* within the code region because they are shared by multiple instructions. Similarly, we sometimes describe registers in the first set as *exclusive* within the code region.

The concepts behind the extension were explored previously using the loop contained in the code fragment shown in Figure 4.7 as an example. The compiler improves the effectiveness of the allocation strategy by selecting instructions that are executed more frequently for inclusion in the set of instructions that are assigned exclusive registers. For example, within region `R2` of the control-flow graph shown in Figure 4.17, basic blocks `BB2` and `BB5` are certain to be executed during every iteration of the loop, whereas only one of `BB3` and `BB4` will be executed during each iteration. Consequently, instructions in basic blocks `BB2` and `BB5` are better candidates for inclusion in the set of instructions assigned to exclusive registers than the instructions in `BB3`. Similarly, when there are enough non-exclusive registers to accommodate basic block `BB4`, so that instruction reuse within region `R1` can be captured in the non-exclusive registers, instructions in basic blocks `BB2` and `BB5` are better candidates for assignment to exclusive registers than the instructions in `BB4`.

Let us consider allocating and scheduling instruction registers within a region that corresponds to a loop that exceeds the instruction register capacity, such as the loop shown in Figure 4.7. The instruction register allocator will partition the instruction registers into two sets. The first set will be used to hold those instructions that remain in the instruction registers for the duration of the loop; the second set will be used to hold those instructions that are loaded during each iteration of the loop. We will denote the number of registers in the first set by r_1 and the number in the second set by r_2.

The compiler will partition the instructions within the allocation region into $n+2$ instruction blocks, where n is a positive integer. The first instruction block will remain in the instruction registers for the duration of the loop; the remaining $n+1$ instruction blocks will be loaded during each iteration of the loop. Thus, the first instruction block may contain up to r_1 instructions, and the remaining $n+1$ instruction blocks may contain up to r_2 instructions. For all but the last region, the compiler must reserve one instruction register for the load operation that transfers the next instruction block in the loop; an instruction register is not needed in the last instruction block in the region because the first instruction block remains in the instruction registers across loop iterations. Thus, the allocator may assume that there are $r_1 - 1$ instruction registers available for instructions in the first instruction block, that there are $r_2 - 1$ instruction registers available for instructions in the next n instruction blocks, and that there are r_2 instruction registers available for instructions in the final instruction block. The compiler must also reserve one instruction register to load the region in the control flow graph that executes after the loop completes. Consequently, if there are R instruction registers available to the allocator, the allocator must choose r_1 and r_2 such that $r_1 + r_2 < R$. Intuitively, the number of instructions that are loaded during each iteration is minimized when the allocator uses all of the available instruction registers,

and the allocator should choose r_1 and r_2 such that $r_1 + r_2 = R - 1$.

The compiler can reduce the number of instructions that are loaded during each iteration of the loop by selecting n and r_2 to minimize

$$(n+1)r_2 \tag{4.1}$$

subject to the constraints

$$r_1 - 1 + n(r_2 - 1) + r_2 = N \tag{4.2}$$

$$r_1 + r_2 = R - 1. \tag{4.3}$$

Equations (4.2) and (4.3) can be used to derive the following equations for computing r_1 and r_2 for a given value of n as follows,

$$r_2 = (N - R + 2)/n + 1 \tag{4.4}$$

$$r_1 = R - r_2 - 1. \tag{4.5}$$

Returning to the loop contained in the code fragment shown in Figure 4.7, we can use the above equations to determine the partitioning of the loop that results in the smallest number of instructions being transferred to the instruction registers. Before instruction register allocation and scheduling, the loop contains 80 instructions. When the allocator is permitted to use all 64 of the instruction registers, we have $N = 80$ and $R = 64$. The values of r_1 and r_2 computed using equations (4.2) and (4.3) for different values of n are tabulated as follows.

n	r_2	r_1	$(n+1)r_2$
1	19	44	38
2	10	53	30
3	7	56	28
4	6	57	30

From the table we can see that the number of instructions transferred is minimized when $n = 3$. This requires partitioning the loop into 5 instruction blocks and allocating 56 instruction registers to those instructions that remain in the instruction registers through the duration of the loop.

Partitioning a region into more instruction blocks increases number of jump-and-load or instruction load operations that need to be scheduled and executed. The compiler may be able to find empty issue slots in which it can schedule these operations without increasing the number of instructions in the region. However, the compiler will sometimes find it necessary to extend the instruction schedule to accommodate the additional instruction register management operations, thereby increasing the code size and execution time. Furthermore, the jump-and-load instructions may introduce additional stall cycles because there is limited opportunity to hide the instruction load latency with other useful operations. Consequently, when allocating and scheduling instruction registers, the compiler attempts to balance performance demands against increased

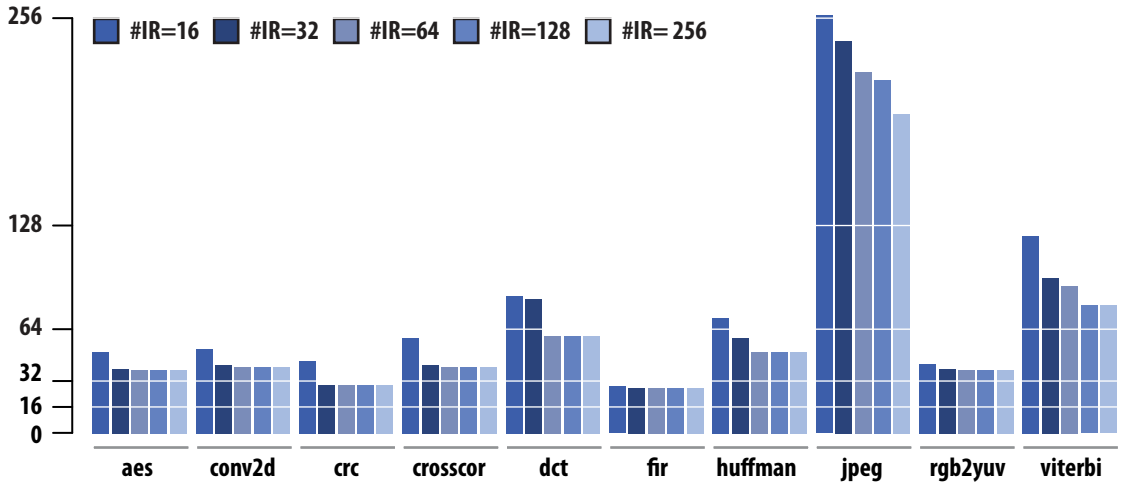


Figure 4.19 – Kernel Code Size After Instruction Register Allocation and Scheduling. The code size decreases as the number of instruction registers that are available to the compiler increases until enough instruction registers are available to capture a kernel.

instruction reuse and the entailing energy-efficiency.

4.5 Evaluation and Analysis

Instruction Registers and Code Sizes

When constructing instruction blocks, the compiler must balance the impact of constructing many small, precise instruction blocks against the impact of instruction load operations on execution times and code sizes. The compiler can reduce the number of instruction loads by constructing and loading larger instruction blocks. Though software controls which instructions are in an instruction block, the compiler may find that expanding an instruction block requires including instructions that may not be executed. For example, the compiler may find that it must include a basic block generated by a conditional control statement to further expand an instruction block. The compiler attempts both to reduce the number of instructions loads that are executed while limiting the number of instructions that are transferred to the instruction registers, though these are sometimes conflicting objectives.

Figure 4.19 shows the size of the kernels after instruction register allocation and scheduling. The size is reported as instruction pairs. To illustrate the impact the number of instruction registers has on code sizes, the figure shows the size of each kernel for different instruction register file capacities. For all of the kernels, 256 instruction registers are enough to accommodate the entire kernels, and adding additional instruction registers does not reduce further the code sizes. Consequently, the 256 instruction register configuration provides an accurate measure of the code size that would result were an infinite number of instruction registers available to the compiler.

As we should expect, code sizes decrease as more instruction registers are made available to the compiler.

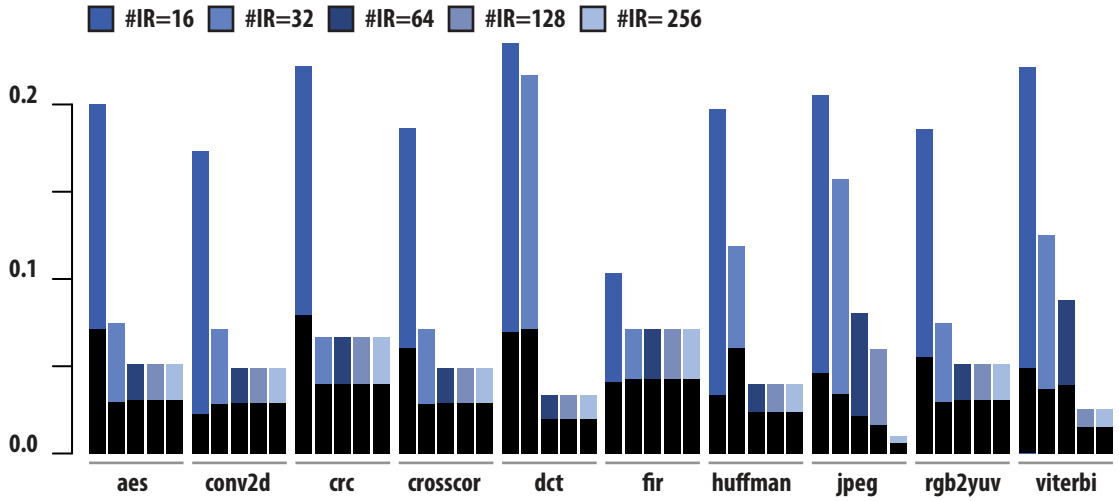


Figure 4.20 – Static Fraction of Instruction Pairs that Load Instructions. The fraction of instruction pairs that contain instruction load and jump-and-load operations is computed by dividing the number of instruction load and jump-and-load instructions by the number of instruction pairs comprising each kernel. The fraction reflects the number of load instructions in the XMU instruction stream. The black component of each bar corresponds to the frequency at which instruction loads appear in the kernel; the remainder of the bar corresponds to jump-and-load instructions.

With more instruction registers, the compiler is able to construct larger instruction blocks and needs to emit fewer instructions to schedule. Once there are enough registers to accommodate the dominant instruction working sets, provisioning additional instruction registers provides small reductions in the static code size.

Figure 4.20 shows the fraction of the instruction pairs that contain instruction load and jump-and-load instructions. The fraction is computed by dividing the number of instruction load and jump-and-load instructions appearing in the code by the number of instruction pairs comprising the kernel. Every kernel has at least one instruction load operation in its prologue to load the body of the kernel, and one jump-and-load instruction its epilogue to transfer control back to the context in which the kernel was invoked. Because 256 instruction registers are sufficient to accommodate any of the kernels, the codes produced for the 256 instruction register configuration have exactly one jump-and-load and one instruction load instruction. The reported fraction measures the number of instruction register management instructions in the XMU instruction stream. Because some of the load instructions are scheduled in what would otherwise be empty XMU issue slots, and because the jump-and-load instructions typically replace jump instructions, we can interpret the data in Figure 4.20 as providing a bound on the increase in code size imposed by having software explicitly manage the instruction registers.

Instruction Registers and Execution Times

Kernel execution times affect both throughput and latency. The normalized kernel execution times are reported in Figure 4.21. To illustrate the impact the number of instruction registers has on execution times, the

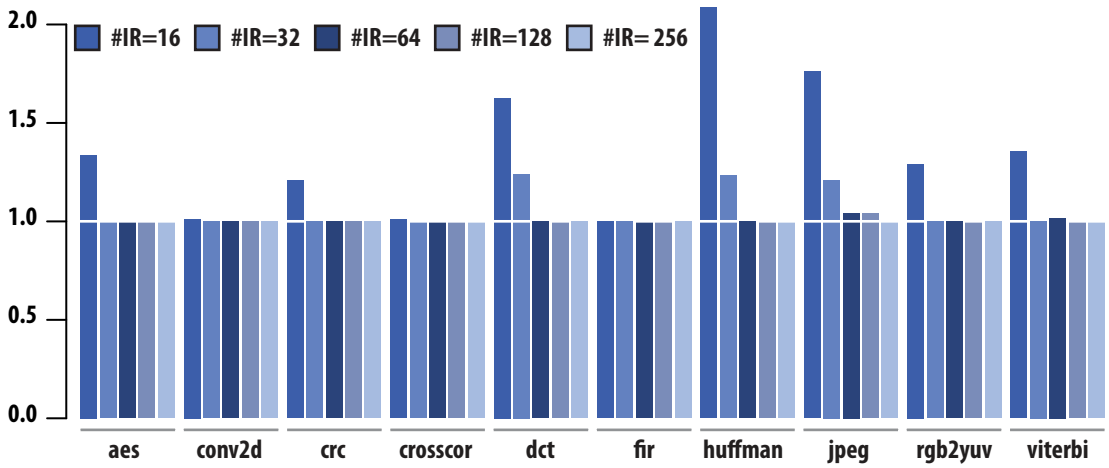


Figure 4.21 – Normalized Kernel Execution Time. The kernel execution time decreases as the number of instruction registers that are available to the compiler increases until there are enough instruction registers to capture a kernel.

figure shows the execution times for different instruction register file capacities.

As we should expect, the execution times decrease as more instruction registers are made available to the compiler. With more instruction registers, fewer instructions need to be loaded during the execution of a kernel, and the processor spends less time stalled waiting for instructions to arrive. The additional instruction registers allow the compiler to construct larger instruction blocks, which reduces the number of instruction load operations that the compiler needs to schedule.

Instruction loads compete with other instructions for XMU issue slots, and may displace compute instructions. Figure 4.22 shows the fraction of the XMU issue slots that are occupied by instruction load and jump-and-load instructions. The reported fraction is computed by dividing the number of instruction load and jump-and-load instructions executed during a kernel by the number of instruction pairs issued during the execution of the kernel.

As Figure 4.22 illustrates, the fraction of issue slots consumed by instruction loads decreases as more instruction registers are made available to the compiler. Intuitively, this results because the compiler is able to construct larger instruction blocks and consequently needs to issue fewer instruction loads. The abrupt reductions in the fraction of issue slots occupied by load instructions we observe when the number of instruction registers exceeds some critical threshold is explained by how the compiler constructions instruction blocks and schedules instruction loads. The compiler attempts to advance instruction loads above loops. Consequently, when the compiler has enough instruction registers available to lift a load out of a loop, the fraction of issue slots occupied by instruction loads decreases by an amount that is proportionate to the contribution of the loop to the execution time.

The data presented in Figure 4.22 explains part of the increase in execution times we observe when we reduce the number of instruction registers and is illustrated in Figure 4.21. Kernels take longer to execute

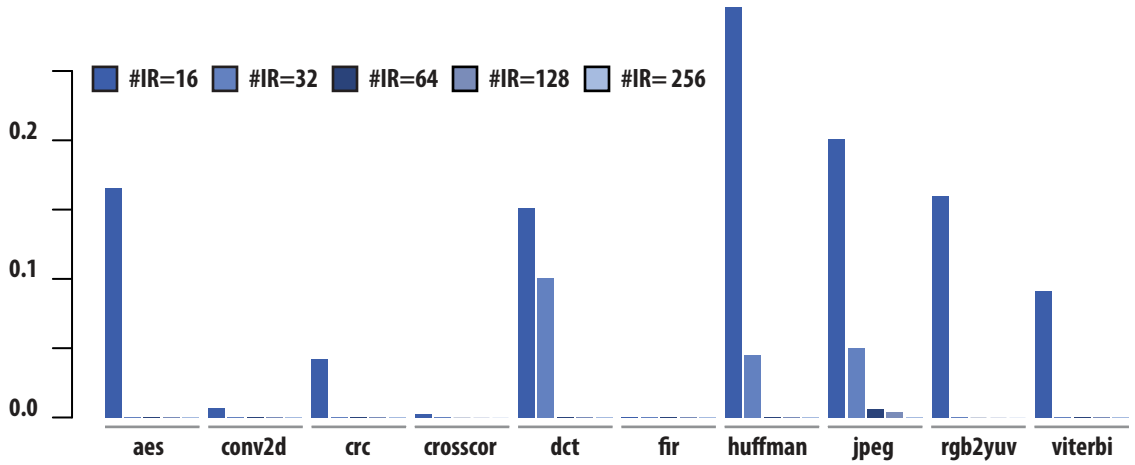


Figure 4.22 – Fraction of Issued Instruction Pairs that Contain Instruction Loads. The reported fraction is computed by dividing the number of instruction load and jump-and-load instructions executed during a kernel by the number of instruction pairs issued during the execution of the kernel.

when there are fewer instruction registers because more time is lost executing load instructions. The remainder of the increase in execution time is due to the processor waiting for instructions to arrive from memory. The processor spends more time waiting for instructions when the instruction register file capacity is reduced because instructions are loaded more often and individual transfers are shorter, providing less time to overlap instruction transfers and computation.

Instruction Bandwidth

Figure 4.23 shows the normalized instruction bandwidth demand at the instruction register file and memory. The normalized instruction bandwidth demand measures the ratio of the instruction bandwidth demand at the instruction register files and the local memory interface. We can interpret it as an estimate of the number of instruction pairs that need to be delivered from the instruction register files and memory each cycle to meet the instruction demands of the processor. The instruction bandwidths are normalized to remove the impact of stalls introduced by data communication. The differences in the instruction bandwidth demands measured for the different kernels and capacities provide insights into the spatial and temporal locality present in the instruction accesses scheduled by the compiler.

As Figure 4.23 illustrates, the normalized instruction bandwidth demand at the instruction register files does not depend on the number of instruction registers, as the demand is determined by the rate at which the processor requests instructions from the instruction registers. However, the additional registers allow the compiler to schedule fewer instruction load operations, and to structure the kernels so that the processor spends fewer cycles stalled waiting for instructions to arrive from memory. The reduction in the number of stall cycles results in an increase in the instruction bandwidth at the instruction register files, as the average number of instruction pairs issued per cycle increases; however, the instruction bandwidth demand remains

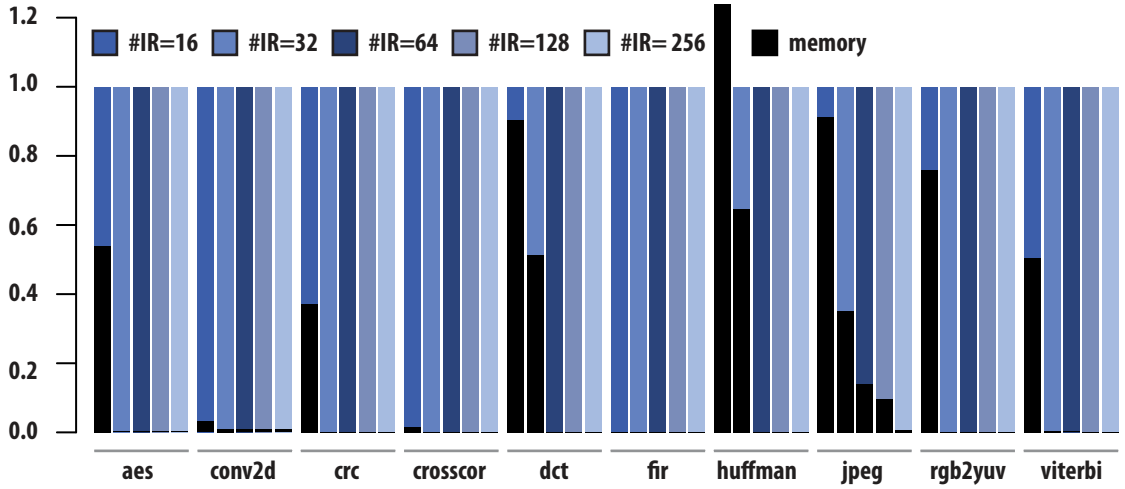


Figure 4.23 – Normalized Instruction Bandwidth Demand. The normalized instruction bandwidth demand measures the ratio of the instruction bandwidth demand at the instruction register files and the local memory interface. The instruction bandwidths are normalized to remove the impact of stalls introduced by data communication.

constant.

The instruction bandwidth demand at the local memory interface decreases as more instruction registers are made available to the compiler, as Figure 4.23 illustrates. Eventually, enough instruction registers are available to capture the loops that dominate the kernels, and the instruction bandwidth demand at the memory interface does not decrease further when additional instruction registers are made available to the compiler. When there are enough instruction registers to capture entire kernels, the instruction bandwidth demand at the memory interface depends only on the number of instructions comprising a kernel and its execution time. Consequently, the instruction bandwidth demand at the memory interface can be made arbitrarily small simply by increasing the number of times a kernel is executed after its instructions are loaded, as this amortizes the initial loading of instructions over more executions of the kernel.

The significant decreases in the instruction bandwidth demand at the memory interface we observe when the additional instruction registers allow the compiler to capture the remainder of an instruction work set of some importance. For example, the loop that accounts for most of the instruction references in the **crc** kernel contains 27 instruction pairs. Increasing the number of instruction registers from 16 to 32 significantly reduces the instruction bandwidth demand at the local memory interface because the additional registers allow the compiler to compile the loop so that instruction loads are not required inside the loop.

The **huffman** kernel is distinct in that the instruction bandwidth demand at the memory interface exceeds the bandwidth demand at the instruction register files for the small instruction register file configurations. This indicates that the number of instructions that are loaded to the instruction registers exceeds the number that are executed, which may happen when software loads instructions that are not executed before being displaced. In the case of the **huffman** kernel, the loop that dominates the execution time and accounts for

the majority of the instruction bandwidth contains several data-dependent control-flow statements within multiple nested loops, which introduce regions of code that are executed during some iterations of the loop but not others. For those regions that contain more than a few instruction pairs, the compiler inserts a short instruction sequence after the control-flow statements that precede the regions to load each region as control enters it. For those regions that contain only a few instruction pairs, the compiler merges each region with the instruction block that contains the control-flow statement, which is more efficient both in terms of execution time and instruction bandwidth. In both cases, the compiler loads some instructions that are not executed during some iterations of the loop; consequently the instruction bandwidth demand at the memory interface exceeds the demand at the instruction register file. Increasing the number of instruction registers allows the instruction registers to capture entire inner loops, which contain enough instruction reuse to reduce the bandwidth demand at the memory interface below the demand at the instruction registers. Though the **jpeg** kernel includes code with similar data-dependent control-flow statements, other parts of the **jpeg** kernel have sufficient instruction reuse to keep the instruction bandwidth demand at the memory interface below the demand at the instruction register file.

Comparison To Filter Caches

This section compares the efficiency of instruction registers to filter caches. Figure 4.24 and Figure 4.25 show the instruction bandwidth at the memory interface for different direct-mapped and fully-associative filter cache configurations. The fully-associative filter cache configurations assume a least-recently-used replacement policy. The instruction bandwidth demand is calculated as the ratio of the number of instructions fetched from memory to number of instructions issued. The bandwidth demand reflects the effectiveness of the filter caches at filter instruction references.

Two trends are immediately apparent in the instruction bandwidth demand data shown in the figures: the instruction bandwidth demand decreases as the filter cache capacity increases; and the instruction bandwidth demand increases as the cache block size increases. The increase in the instruction bandwidth demand we observe when the cache block size increases deserves some attention because we would generally prefer to use a large cache block size, as increasing the cache block size reduces the number of tags that are required to cover the filter cache entries and reduces the memory access time penalty imposed by cache misses. Figure 4.2 illustrates the impact of the cache block size on the number of instructions that must be fetched from memory and the miss rate.

Increasing the cache block size may introduce additional conflict misses in codes with loops that are similar in size to the capacity of the filter cache, as instructions at the start and end of the loop may map to the same entry when the block size is increased. Typically, this occurs when the loop is not aligned to cache block boundaries, which has the effect of reducing the effective capacity of the filter cache. The **crc** kernel contains an inner loop that exhibits this problem for the smallest configurations when the cache block size increases from 4 instruction pairs to 8 instruction pairs. The compiler may ameliorate the problem by aligning all loops to cache block boundaries, though this increases the size of the kernel in the backing memory and may introduce additional capacity misses.

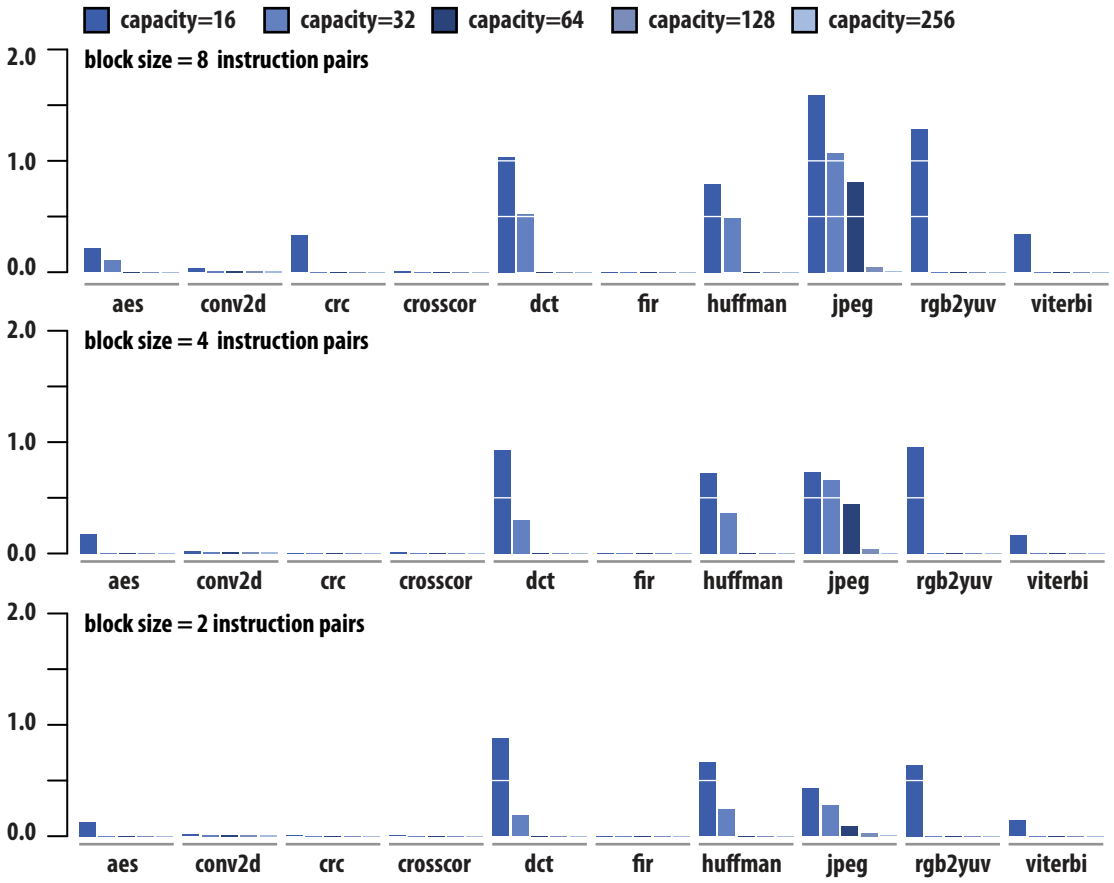


Figure 4.24 – Direct-Mapped Filter Cache Bandwidth Demand. The instruction bandwidth demand is calculated as the ratio of the number of instructions fetched from memory to number of instructions issued.

Increasing the cache block size invariably results in some instructions that will not be executed being unnecessarily transferred to the cache when servicing a cache miss. For example, instructions following a control-flow instruction and residing in the same cache block will be transferred with the control-flow instruction even though the subsequent instructions will not be executed when the control-flow instruction transfers control to an instruction residing in a different cache block. This effect accounts for part of the steady increase in the instruction bandwidth demand we observed for the **huffman** and **jpeg** kernels when the block size increases.

Filter caches and instruction registers improve energy efficiency by filtering instruction references that would otherwise reach more expensive memories. The instruction bandwidth demand at the memory interface provides a measure of how effective the different filter cache and instruction register configurations are at filtering instruction references. However, we cannot directly compare the miss rates to evaluate relative improvements in energy efficiency because the instruction streams differ. Instead, we need to compare the

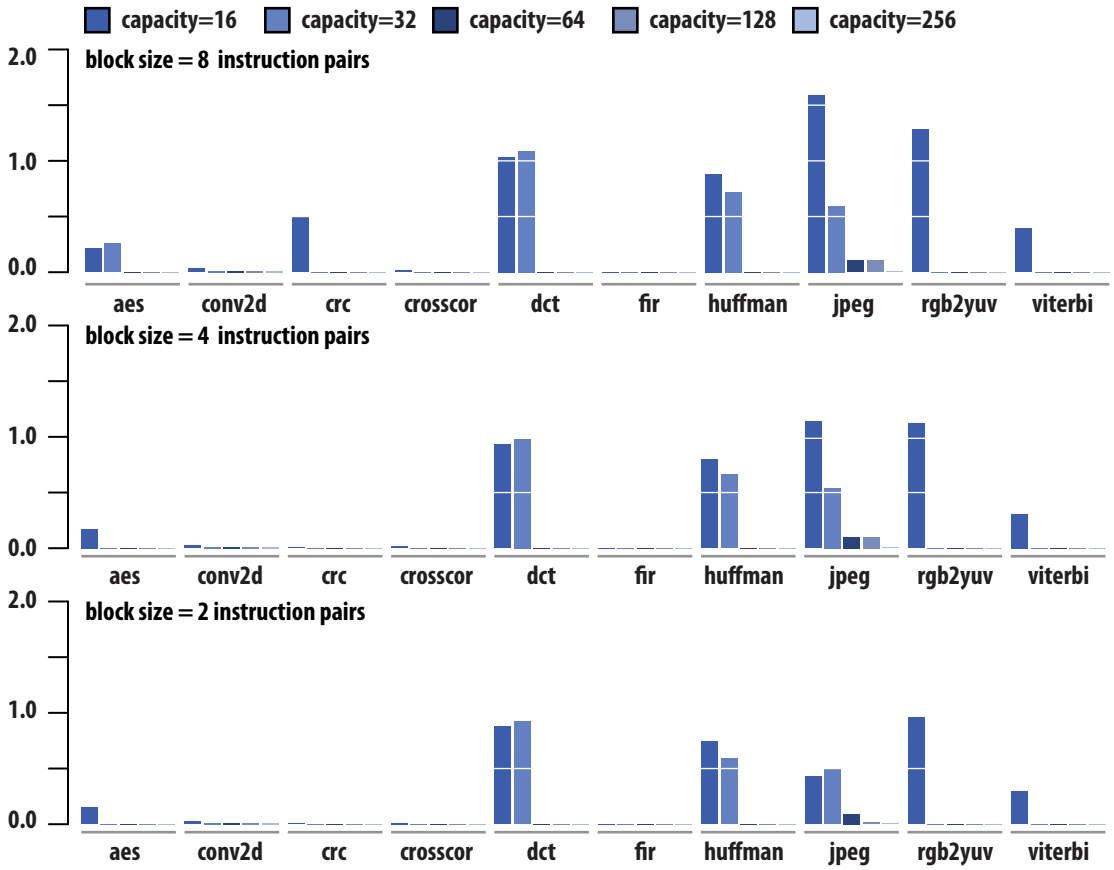


Figure 4.25 – Fully-Associative Filter Cache Bandwidth Demand. The instruction bandwidth demand is calculated as the ratio of the number of instructions fetched from memory to number of instructions issued. The filter cache controller uses a least-recently-used replacement policy.

relative number of instructions fetched from memory, which is shown in Figure 4.26. The filter cache configurations use a cache block size of 2 instruction pairs, and correspond to the configurations with the smallest instruction bandwidth demands at the capacities shown.

As the data illustrated in Figure 4.26 demonstrates, the number of instructions fetched from memory is typically a small fraction of the number of instructions executed by most kernels. The notable exception is the fully-associative filter cache with 32 entries, which performs poorly on the **dct** kernel. The **dct** kernel is dominated by an inner loop that contains 38 instruction pairs. The least-recently-used replacement policy results in the cache controller replacing every instruction stored in the cache during each execution of the loop. Similar adverse interactions between working sets and the replacement policy also explains the behavior of the fully-associated filter cache for the **huffman** and **jpeg** kernels. For those kernels where the instruction register configuration fetches more instructions than the direct-mapped filter cache, the preponderance of the additional instructions loaded from memory are introduced by the compiler to manage the instruction

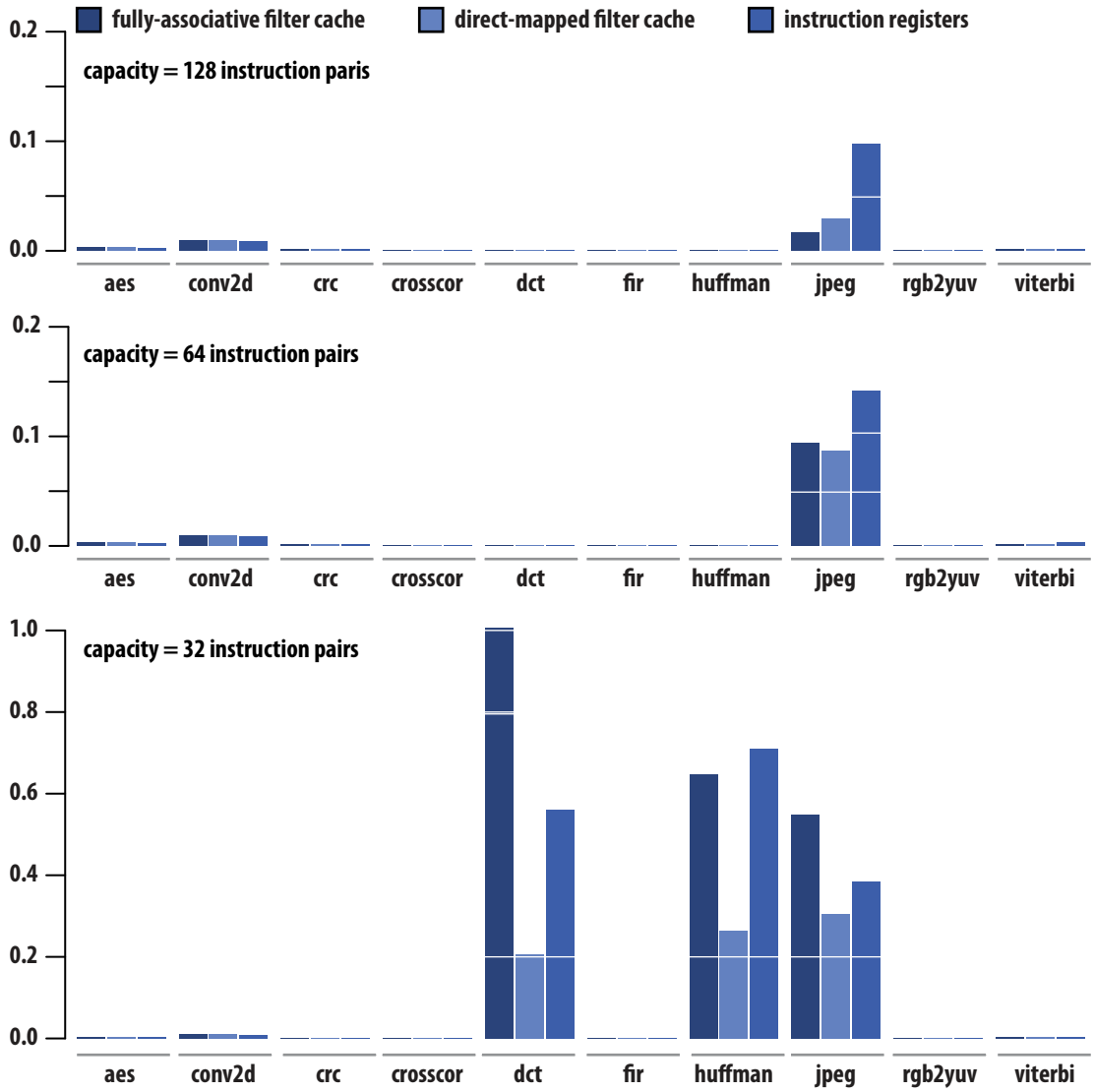


Figure 4.26 – **Comparison of Instructions Loaded Using Filter Caches and Instruction Registers.** The number of instructions loaded is shown normalized to the number of instructions executed in the filter cache configurations. The filter caches have a block size of 2 instruction pairs. The instruction register management instructions increase the dynamic instruction count in the instruction register configurations, which contributes to the increased number of instructions that are loaded from memory.

registers. We should note that the number of instructions loaded from memory for the **conv2d** kernel is lower for the instruction register configurations than the filter cache configurations, despite the presence of the instruction register management instructions in the instruction stream. The compiler is able to achieve this by partitioning the instruction registers as described in the previous section.

Arithmetic Operations	Typical Energy	Relative Energy
32-bit addition	520 fJ	1×
16-bit multiply	2,200 fJ	4.2×
Instruction Cache — 2048 Instruction Pairs [Direct-Mapped]		
Control Flow Logic	445 fJ	0.85×
Instruction Pair Fetch	18,000 fJ	34.6×
Filter Cache — 64 Instruction Pairs [Direct-Mapped]		
Control Flow Logic	445 fJ	0.85×
Instruction Pair Fetch	1,000 fJ	1.9×
Instruction Pair Miss	1,000 fJ	1.9×
Instruction Pair Refill [2 Pairs]	1,500 fJ	2.9×
Filter Cache — 64 Instruction Pairs [Fully-Associative]		
Control Flow Logic	445 fJ	0.85×
Instruction Pair Fetch	1,400 fJ	2.7×
Instruction Pair Miss	650 fJ	1.2×
Instruction Pair Refill [2 Pairs]	1,600 fJ	3.0×
Instruction Registers — 64 Instruction Pairs		
Control Flow Logic	154 fJ	0.3×
Instruction Pair Fetch	330 fJ	0.6×
Instruction Pair Refill	1,160 fJ	2.2×

Table 4.1 – Elm Instruction Fetch Energy. The data were derived from circuits designed in a 45 nm CMOS process that is tailored for low standby-power applications. The process uses thick gate oxides to reduce leakage, and a nominal 1.1 V supply.

Table 4.1 lists the typical energy expended fetching instructions from the different instruction stores. The energy consumed performing common arithmetic operations is included for comparison. The data presented in the table were determined from detailed transistor-level simulations of the memory and logic circuits comprising the different stores. The circuit models were extracted after layout, and include parasitic device and interconnect capacitances.

The filter caches use a cache block size of 128 bits, which accommodates 2 instruction pairs. The direct-mapped filter cache uses an SRAM array for tag storage. The tag and instruction arrays are accessed in parallel. Consequently, the energy expended accessing the cache does not depend on whether the access hits or misses. The fully-associative filter cache uses a CAM array for tag storage. The word-line of the instruction array is driven by the match line from the CAM array so that the instruction array is only accessed when there is a match. Consequently, the energy expended accessing the fully-associative cache depends on whether there is a hit, with more energy expended on a hit than a miss. The refill energy values include the energy expended writing both the tag and instruction array. The energy expended accessing the backing instruction cache and transferring the instructions to the filter cache is included in the fetch entry for the instruction cache.

Figure 4.27 shows the energy consumed delivering instructions to the processor when executing the benchmark kernels, from which we can compare the energy efficiencies of the different architectures. To account for differences in the dynamic instruction counts, the energy reported is the aggregate energy consumed delivering instructions through the execution of the kernel. The capacity of the instruction registers and

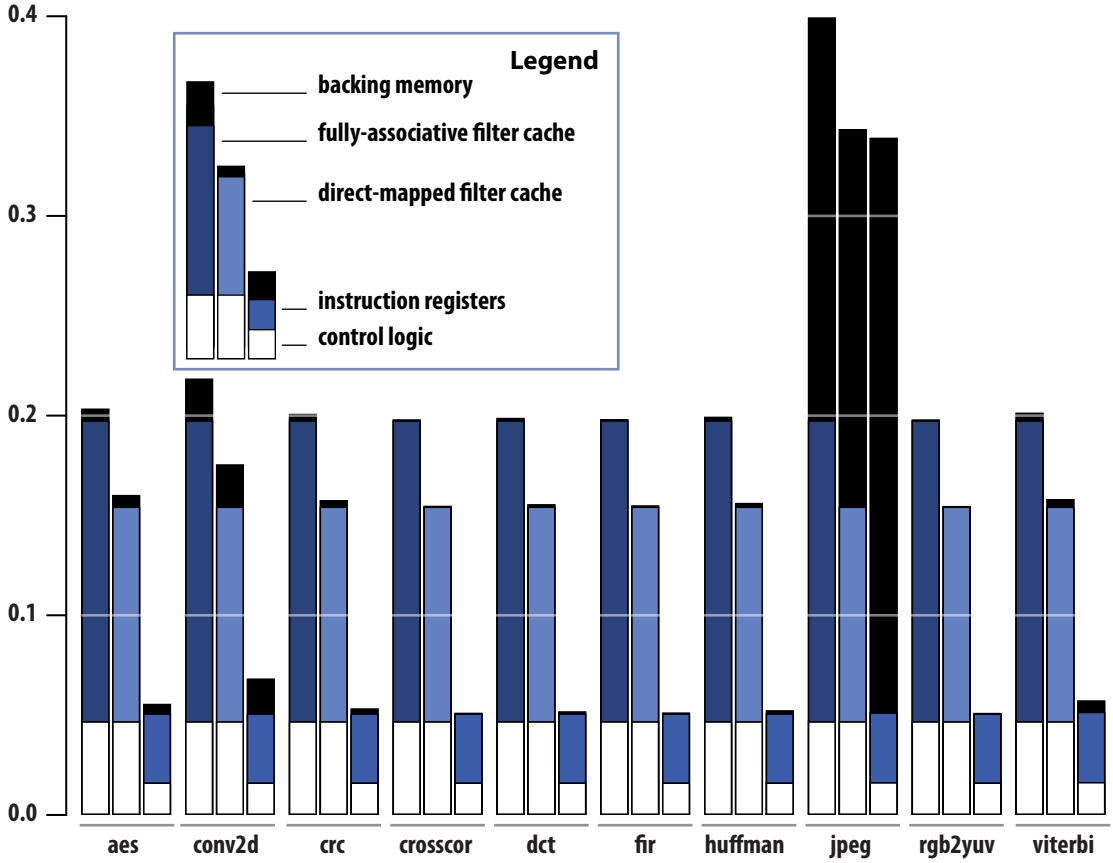


Figure 4.27 – **Instruction Delivery Energy.** The capacity of the instruction registers and filter caches for the configurations shown is 64 instruction pairs; the local memory that backs the instruction registers and filter caches has a capacity of 1024 instruction pairs. The energy shown is normalized to the energy consumed that would be consumed delivering instructions directly from the local memory.

filter caches for the configurations shown is 64 instruction pairs; the local memory that backs the instruction registers and filter caches has a capacity of 1024 instruction pairs.

To illustrate the improvement in energy efficiency the different architectures achieve, the energy shown is normalized to the energy consumed delivering instructions directly from the local memory.

The control logic component of the energy includes the program counter register, the arithmetic circuits that update the program counter and compute branch target addresses, and the control logic and multiplexers that select the program counter value used to fetch instructions. The instruction register component of the energy includes the energy expended writing instructions to the instruction registers, reading instructions from the instruction registers, and transferring instructions to the data-path. Similarly, the filter cache component of the energy includes the energy expended reading and writing instructions to the filter caches, reading the tag entries, and transferring instructions to the data-path. We have not included the energy expended in the cache controllers required for the filter caches, and consequently the energy reported for the filter cache

configurations provides a lower bound. The backing memory component of the energy includes the energy expended reading instructions from the local memory and transferring instructions to the instruction registers or filter caches.

Because the different architectures transfer different numbers of instructions from the backing memory, the relative energy efficiencies of the different architectures is sensitive to the cost of accessing the backing memory. This may make instruction register organizations appear less efficient, particularly when the instructions that are required to manage the instruction registers contribute to instruction working sets that exceed the capacity of the instruction registers. Were the amount of energy expended accessing the backing memory to increase, the instruction register organization would appear less efficient when executing the **jpeg** kernel because it requires that a significant number of instructions be fetched from the backing memory. However, the **jpeg** kernel can be partitioned into multiple smaller kernels with commensurately smaller working sets similar to the **dct** and **huffman** kernels, which are readily captured in the instruction registers.

4.6 Related Work

Unlike reactive mechanisms such as filter caches [82], software-managed instruction registers reduce instruction movement and avoid the performance penalties caused by high miss rates. Dynamically loaded loop caches [95], which are loaded on short backwards branches, do not impose miss penalties, but cannot capture loops containing control flow and execute the first two iterations of each loop out of the backing instruction cache while the loop cache is loaded. Pre-loaded loop caches [50] eliminate tag checks when executing instructions in the loop cache, but cannot be dynamically loaded: they must be large to capture any significant fraction of an application's instructions and require hardware to detect when instructions should be issued from the loop cache.

The instruction register file proposed by Hines, Whalley, and Tyson [64, 65] provides a form of code compression in which the most frequently encountered instructions are issued from a small register file after a packed sequencing instruction is fetched from the instruction cache; the registers cannot be dynamically loaded. The tagless instruction cache described by Hines, Whalley, and Tyson [66] uses a clever scheme for modifying a conventional filter cache to exploit situations in which the tag comparison is certain to produce a match. A common example of an access that is certain to produce a match is fetching consecutive instructions from within the same cache block. This observation is commonly exploited by instruction buffers [29] and cache line buffers [45]. Effectively, the tagless cache is able to operate like an extended line buffer. The scheme is not truly tagless, as a truncated identifier is used in place of a conventional tag. The scheme requires a complicated cache controller, and replaces the filter cache tags with auxiliary data that the controller uses to determine whether an instruction is present in the tagless cache. Those configurations that perform best require more auxiliary information than would be required to store the tags used in a conventional filter cache.

The explicit data-flow graph execution model implemented in the TRIPS processor prototype defines a block-atomic execution model [107] that is similar to earlier block-atomic architectures [103]. The block-atomic execution model implemented in the TRIPS processor executes groups of instructions as atomic units. Instruction groups are fetched, executed, and committed as an atomic unit. Any architectural effects of the

instructions in an atomic block may be observed by instructions in different atomic instruction blocks only after all of the instruction in the block have been executed and are ready to commit. The compiler maps instruction groups into predicated hyperblocks. A hyperblock has a single point of entry and may have multiple points of exit. The execution model does not allow control to transfer within a hyperblock. Encoding instruction groups as hyperblocks reduces the area and complexity of the branch and branch target prediction hardware implemented in the TRIPS track hyperblocks, as the hardware structures maintain information at the granularity of hyperblocks rather than individual instructions. The TRIPS processor prototype uses fixed hyperblock size defined by hardware, which results in significant overhead, as the number of instructions available in an instruction group is often insufficient to completely fill a hyperblock. Using large atomic instruction blocks reduces the cost and complexity of branch prediction. Information used for branch prediction is maintained at the block level rather than the instruction level, so structure with a fixed number of entries is able to cover more of an application. However, effectiveness of this technique is partially offset by the expanded code size that results from using a fixed hyperblock size [44].

The vector-thread execution model uses atomic instruction blocks to allow software to encode spatial and temporal locality in the instruction streams executed by the virtual processors [88]. The virtual processors execute RISC-like instructions that are grouped into atomic instruction blocks. An atomic instruction block has a single point of entry and may have multiple points of exit. Software specifies the number of instructions contained in an atomic instruction block. An atomic instruction block may be issued to the virtual processors by the control processor, or may be explicitly fetched by a virtual processor. An atomic instruction block may not contain internal transfers of control, as there is no notion of a program counter or implicit fetch mechanism. Instead, atomic instruction blocks must be explicitly issued by the control processor or fetched by a virtual processor. Because there is no notion of a program counter, the virtual processors can use a short instruction pointer to record the position within an atomic instruction block of the next instruction to be executed. The atomic instruction block abstraction allows the control processor to issue a large number of instructions to the virtual processors as a single operation.

The MIT Scale prototype implementation of the vector-thread model distributes atomic instruction block caches close to the execution clusters [88]. Distributing the cache memories among the execution clusters allows instructions to be cached close to the function units. Instruction fetch commands specify the address of the atomic instruction block to be fetched. The specified address is compared against a set of tags to determine whether the instructions in the atomic instruction block are present in the cache. If there is a miss, an instruction fill unit fetches the atomic instruction block from memory and stores it to the cache. Grouping instructions into atomic instruction blocks allows a single tag comparison to determine whether all of the instructions in the block are present. Though Scale does not allow control to transfer within an atomic instruction block, instructions may be predicated, or control may conditionally transfer to a different atomic block. Scale's vector execution model allows Scale to eliminate many of the control overheads associated with executing data-parallel loops, and the vector execution model further reduces the number of instruction tag comparisons that the processor must perform when executing vectorizable codes. However, supporting an execution model in which a large vector of virtual processors are mapped onto a much smaller number of physical processors imposes considerable hardware complexity.

The block-aware instruction set architecture described by Zmily and Kozyrakis uses basic block descriptors to describe the block structure of program codes to hardware [157]. This allows the processor front-end to decouple control-flow speculation from the fetching of instructions stored in the instruction cache, which improves the performance and energy-efficiency of superscalar processors. The proposed block-aware architecture defines basic block descriptors that software uses to describe the structure of the basic blocks comprising a program. Descriptors specify information such as the size of a basic block, the type of control-flow instruction that terminates the basic block, and hints provided by the compiler for predicting the direction of conditional branches. The basic block descriptors are stored in ancillary structures to decouple control-flow speculation from the fetching and decoding of instructions from the instruction cache. The descriptors are used to improve the accuracy of instruction prefetching. Because descriptors can be fetched before control transfers to the basic blocks they describe, the front-end can tolerate longer instruction cache access times. This allows microarchitecture optimizations that improve energy efficiency to be employed in the front-end. For example, the instruction cache can be designed so that the instruction array is accessed after the tag comparison has been completed to avoid accessing the memory array on a miss.

4.7 Chapter Summary

Instruction registers provide distributed collections of software-managed registers that extend the memory hierarchy used to deliver instructions to function units. They allow critical instruction working sets such as loop bodies and computation kernels to be stored close to function units in registers that are inexpensive to access. Instruction registers are efficiently implemented as distributed collections of small register files. These small register files are fast and inexpensive to access, and distributing them allows instructions to be stored close to function units, reducing the energy consumed transferring instructions to data-path control points.

Chapter 5

Operand Registers and Explicit Operand Forwarding

This chapter introduces operand registers and explicit operand forwarding, both of which reduce the energy consumed delivering operands to a processor's function units. Operand registers and explicit operand forwarding let software capture short-term data reuse and instruction-level producer-consumer data locality in inexpensive registers that are closely integrated with the function units in a processor. Both operand registers and explicit forwarding expose to software the spatial distribution of registers among function units, which increases software control over the movement of data between function units and allows software to place data close to where they will be consumed, reducing the energy expended transporting operands to function units. Operand registers and explicit operand forwarding allow the compiler to keep a significant fraction of the data bandwidth within the kernels of embedded applications close to the function units, which reduces the amount of energy expended staging data in registers.

The remainder of this chapter is organized as follows. I begin by introducing the basic concepts motivating explicit operand forwarding and operand registers. I then describe how explicit operand forwarding and operand registers extend a conventional register organization and affect instruction set architecture, using Elm as a specific example of a machine that uses both explicit operand forwarding and operand registers. I then discuss microarchitecture, again using Elm as an example of an architecture that implements explicit operand forwarding and operand registers. After describing the architecture, I discuss compilation issues such as instruction scheduling and register allocation. An evaluation follows. I conclude with related work and then summarize the chapter.

5.1 Concepts

Explicit operand forwarding allows software to control the routing of data through the forwarding network so that ephemeral values can be delivered directly from pipeline registers. This lets software avoid the expense of storing ephemeral values to register files when ephemeral values can be consumed while they are available

in the forwarding network.

Operand registers extend a conventional register organization with distributed collections of small, inexpensive general-purpose operand register files, each of which is integrated with a single function unit in the execute stage of the pipeline. Essentially, operand registers extend the pipeline registers that deliver operands to function units in a conventional pipeline into shallow register files; these shallow operand register files retain the low access energy of pipeline registers while capturing a greater fraction of operand references. This lets software capture short-term reuse and instruction-level producer-consumer locality near the function units. Both explicit operand forwarding and operand registers increase software control over the movement of operands between function units and register files.

Explicit Operand Forwarding

Explicit operand forwarding lets software control the routing of operands through the forwarding network. Consider the following code fragment.

```
int x = a + b - c;
```

During compilation, the compiler translates statements that involve multiple operations into sequence of primitive operations, each of which corresponds directly to a machine instructions or a short, fixed-sequence of machine instructions. Most compilers would translate the above code fragment into a sequence of operations similar to the following.

```
int t = a + b;
int x = t - c;
```

The compiler introduces the temporary variable `t` to forward the result of the addition operation to the subtraction operation. The corresponding sequence of machine instructions produced by the compiler after performing register allocation and instruction selection would be similar to the following sequence of assembly instructions, though the register identifiers might differ.

```
add r5 r2 r3;
sub r1 r5 r4;
```

When the above sequence of instructions executes, the value of the compiler-introduced temporary `t` assigned to register `r5` is delivered by the forwarding network. Register `r5` is unnecessarily written when the instructions execute. Register `r5` simply provides an identifier that allows the logic that controls the forwarding network to detect that the result of the `add` instruction must be forwarded to the `sub` instruction. Writing the result of the addition to the register file consumes more energy than performing the addition. For example, performing a 32-bit addition in a 45 nm low-power CMOS technology consumes about 0.52 pJ; writing the result of the addition to a 32-entry register file consumes about 1.45 pJ. Furthermore, using register `r5` to forward the intermediate value unnecessarily increases register pressure, as whatever value was previously in register `r5` is lost after the `add` instruction completes.

Explicit operand forwarding lets software orchestrate the routing of operands through the forwarding network so that ephemeral values need not be written to register files. Instead, ephemeral values can be consumed while they are available in the forwarding network. Explicit operand forwarding introduces the concept of *forwarding registers* to provide an explicit architectural name for the result of the last instruction that a function unit executed. Forwarding registers establish architectural names for the physical pipeline registers residing at the end of the execute (EX) stage. Software can use forwarding registers as operands to explicitly forward the result of a previous instruction. Similarly, software can use a forwarding register as a destination to direct the result of an instruction to a forwarding register, which indicates that the result should not be written to the register file. For example, the following sequence of assembly instructions forwards the result of the `add` operation through forwarding register `t0`, which is the forwarding register name for the pipeline register that captures the output of the ALU.

```
add t0 r2 r3;
sub r1 t0 r4;
```

Forwarding register `t0` appears in two different contexts: it appears as the destination of the `add` instruction to specify that the result of the addition does not need to be stored in the register file, and it appears as an operand to the `sub` instruction to specify that the fist operand of the subtraction operation is the result produced by the previous instruction.

Operand Registers

Registers provide low-latency and high-bandwidth locations for storing temporary and intermediate values, and for staging data transfers between memory and function units. Registers are usually implemented as multi-ported register files, which are denser and faster than latches and flip-flops, though not as dense as SRAM. Most of the architectural registers that are significant to software are implemented as registers in register files. Some registers that are accessed every cycle, such as the program counter, or are accessed irregularly, such as status registers that are accessed through special instructions, are implemented independently as dedicated registers.

Accessing data stored in registers is considerably less expensive than accessing data stored in memory, both in terms of execution time and energy. Consequently, optimizing compilers spend considerable effort allocating registers in order to limit the number of expensive memory accesses that are performed. Increasing the number of registers that are available to software allows more data to be stored in registers rather than memory, reducing the number of memory references. Similarly, increasing the number of registers that are exposed to the compiler allows larger working sets to be captured in registers, reducing the number of variables that must be spilled to memory when register demand exceeds register availability. In general, the ability of the register file to filter references depends on its capacity relative to the working sets present in an application. However, providing more registers requires additional register file capacity. Because larger register files are more expensive to access, the cost of accessing data stored in registers tends to increase as more registers are made available. Ignoring the complexity of encoding larger register specifiers in a fixed-length instruction word, there is a fundamental tension between the benefits and costs of increasing the

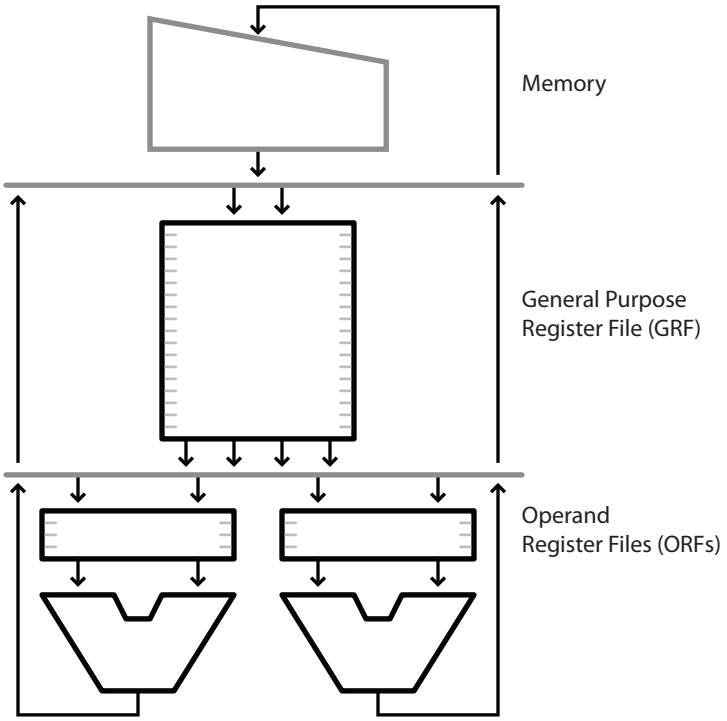


Figure 5.1 – **Operand Register Organization.** The distributed and hierarchical register organization lets software exploit reuse and locality to deliver a significant fraction of operands from registers that are close to the function units. The distributed operand register files form the first level of the register hierarchy, and are used to store small, intensive working sets close to the function units. The general-purpose register file forms the second level of the register hierarchy, and is used to store larger working sets. A function unit may receive its operands from its local operand register file or the general-purpose register file, and it may write its result to any of the register files. Operand registers let software capture short-term reuse and instruction-level producer-consumer locality close to the function units, which keeps a significant fraction of the operand bandwidth local to each function unit. The capacity of the general-purpose register file allows it to capture larger working sets, exploiting reuse and locality over longer intervals than can be captured in the small operand register files. Reference filtering by the operand register files reduces demand for operand bandwidth at the general-purpose register file.

capacity of register files to capture larger working sets. The additional capacity allows more temporary and intermediate values to be stored in registers rather than memory, but makes register accesses more expensive.

Operand registers extend a conventional register organization with small collections of registers that are distributed among the function. Figure 5.1 illustrates a register organization with operand registers. Like explicit forwarding, operand registers let software capture short-term reuse and instruction-level producer-consumer locality within the kernels of embedded applications close to the function units, and they reduce the number of accesses to the larger general-purpose register file. However, unlike forwarding registers, operand registers are architectural and are preserved during exceptions. Operand registers are more flexible than forwarding registers, and are able to capture reuse and instruction-level producer-consumer locality

within longer instruction sequences. This allows operand registers to capture a greater fraction of operand bandwidth and to offer more relaxed instruction scheduling constraints.

Like explicit operand forwarding, operand registers exploit short-term reuse and locality to reduce demand for operand bandwidth at the general-purpose register. The operand registers effectively filter data references closer to the function units than the general-purpose registers. This keeps more communication local to the function units and improves energy efficiency: the small operand register files are inexpensive to access, and the expense of traversing the forwarding network and pipeline register preceding the execute (EX) stage of the pipeline is avoided. For example, delivering two operands from a general-purpose register file and writing back the result consumes 3.1 pJ, which exceeds the 2.2 pJ consumed by a 16-bit multiplication; using operand registers to stage the operands and the result consumes only 0.78 pJ, which is close to the 0.52 pJ consumed by a 32-bit addition.

5.2 Microarchitecture

Forwarding registers establish architectural names for the physical pipeline registers residing at the end of the execute (EX) stage. Forwarding registers retain the result of the last instruction that wrote the execute (EX) pipeline register. To preserve the registers, data-path control logic clock-gates the pipeline registers during interlocks and when **nop**s execute. This allows operands to be explicitly forwarded between instructions that execute multiple cycles apart when intervening instructions do not update the physical registers used to communicate the operands [142].

The value returned from the local memory when a load instruction executes cannot be explicitly forwarded because load values do not traverse the pipeline registers at the end of the execute (EX) stage; instead, architectural registers retime load values arriving from memory.

Processors with deeper pipelines generally need more registers to cover longer operation latencies. Explicit forwarding can be extended for deeper pipelines by establishing additional explicit names for the results of recently executed instructions that are available within the forwarding network. However, the pipeline registers that stage the forwarded operands are not conventional architectural registers and may be difficult to preserve when interrupts and exceptions occur. When interrupts are a concern, the compiler can be directed to disallow the explicit forwarding of operands between instructions that may cause exceptions.

Operand Registers

Operand registers are implemented as very small register files that are integrated with the function units in the execute stage of the pipeline. This allows operand registers to provide the energy efficiency of explicit forwarding while preserving the behavior of general-purpose registers. Accessing the operand register files in the execute (EX) stage may contribute to longer critical path delays. However, the operand register read occurs in parallel with the activation of the forwarding network, and much of the operand register file access time is covered by the forwarding of operands from the write-back (WB) stage. Regardless, register organizations using operand registers require fast, and consequently small, operand register files to avoid creating

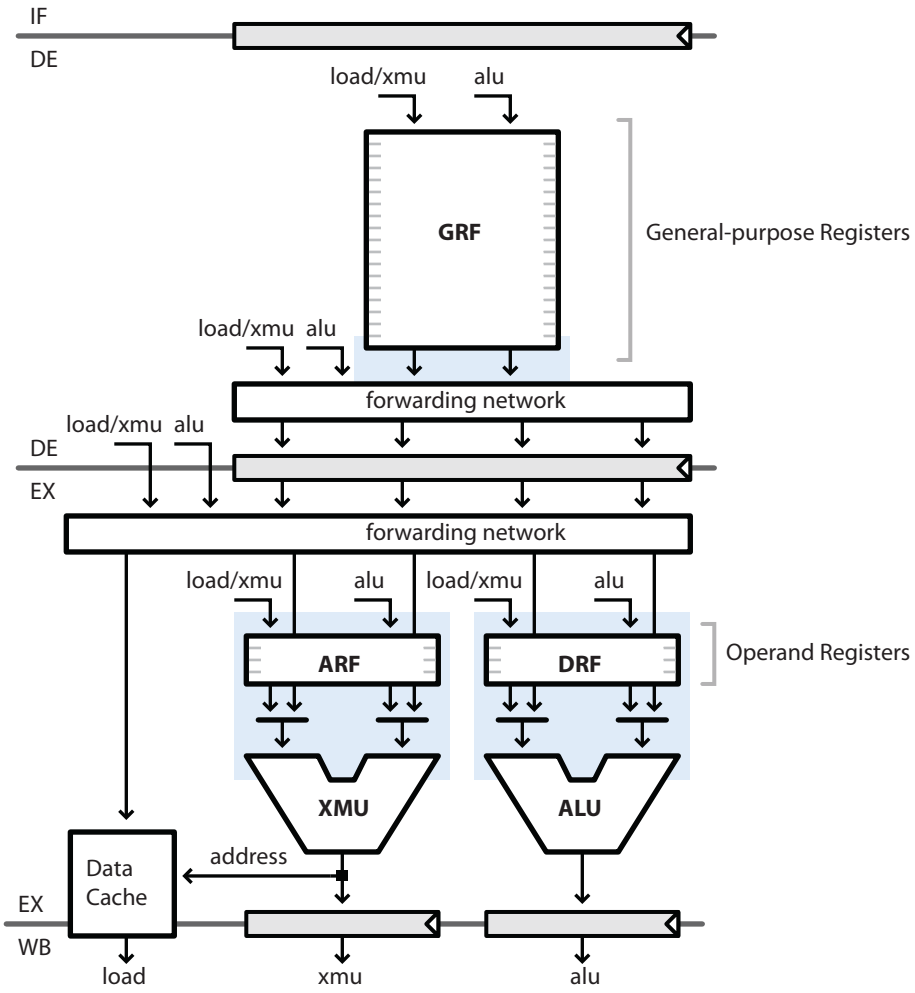


Figure 5.2 – **Operand Register Microarchitecture.** The operand register files precede the function units in the execute (EX) stage of the pipeline. The extended-memory unit (XMU) performs memory operations and simple arithmetic operations. The arithmetic and logic unit (ALU) performs simple and complex arithmetic instructions. The operand register file associated with the extended-memory unit is referred to as the address register file (ARF), and the operand register file associated with the arithmetic and logic unit is referred to as the data register file (DRF). The operand register files are backed by the deeper general-purpose register file (GRF). Reference filtering reduces demand for operand bandwidth at the general-purpose register file, reducing the number of read ports needed at the general purpose register file.

critical paths in the execute stage.

Each operand register is assigned to a single function unit, and only its associated function unit can read it. The function units can write any register, which allows the function units to communicate through operand registers or the more expensive backing registers. This organization allows software to keep a significant fraction of the operand bandwidth local to the function units, to place operands close to the function unit that will consume them, and to keep data produced and consumed on the same function unit local to the function

Arithmetic Operations		Relative Energy
32-bit addition	520 fJ	1×
16-bit multiply	2,200 fJ	4.2×
General-Purpose Register File — 32 words (4R+2W)		
32-bit read	830 fJ	1.6×
32-bit write back	1,450fJ	2.8×
General-Purpose Register File — 32 words (2R+2W)		
32-bit read	800 fJ	1.5×
32-bit write back	1,380 fJ	2.7×
Operand Register File — 4 words (2R+2W)		
32-bit read	120 fJ	0.2×
32-bit write back	540 fJ	1.0×
Forwarding Register — 1 word (1R+1W)		
32-bit forward	530 fJ	1.0×
Data Cache — 2K words (1R+1W) [8-way set-associative with CAM tags]		
32-bit load	10,100 fJ	19.4×
32-bit store	14,900 fJ	28.6×

Table 5.1 – Energy Consumed by Common Operations. The data were derived from circuits implemented in a 45 nm CMOS process that is tailored for low standby-power applications. The process uses thick gate oxides to reduce leakage, and a nominal supply of 1.1 V to provide adequate drive current.

unit. Furthermore, reference filtering by the operand registers reduces demand for operand bandwidth from the shared general-purpose registers, which allows the number of read ports to the general-purpose register file to be reduced without adversely impacting performance. This makes the general-purpose register file smaller and shortens its bit-lines, reducing its access costs by decreasing the bit-line capacitance switched during read and write operations. It also reduces the cost of updating a register because fewer access devices contribute to the capacitance internal to each register.

5.3 Example

Consider the following code fragment, which computes the product of the complex values `a` and `b` and assigns the product to `x`. The operands are 16-bit fixed-point integers.

```
100  x.re = a.re * b.re - a.im * b.im;
101  x.im = a.re * b.im + a.im * b.re;
```

Table 5.1 lists typical amounts of energy expended performing the arithmetic operations, memory operations, and register file accesses performed in the example code. The register file read energy includes the cost of transferring an operand from the register file to an input of a function unit. The register file write energy includes the cost of transferring a result from the output of a function unit to the register file.

Figure 5.3 shows the sequence of operations generated by the compiler, and Figure 5.4 shows the corresponding sequence of assembly instructions for a conventional register organization. The assembly listing assumes that the address of variable `a` resides in register `r10`, the address of `b` resides in `r11`, and the address

```
// source                                    // compiled

100  x.re = a.re * b.re - a.im * b.im;       200  t1  = load [a.re];
101  x.im = a.re * b.im + a.im * b.re;       201  t2  = load [a.im];
                                             202  t3  = load [b.re];
                                             203  t4  = load [b.im];
                                             204  t5  = mult t1 * t3;
                                             205  t6  = mult t2 * t4;
                                             206  t7  = sub  t5 - t6;
                                             207  [x.re] = store t7;
                                             208  t8  = mult t1 * t4;
                                             209  t9  = mult t2 * t3;
                                             210  t10 = add t8 + t9;
                                             211  [x.im] = store t10;
```

Figure 5.3 – **Intermediate Representation.**

```
// source                                    // assembly

100  x.re = a.re * b.re - a.im * b.im;       200  ld r1 [r10+0];
101  x.im = a.re * b.im + a.im * b.re;       201  ld r2 [r10+1];
                                             202  ld r3 [r11+0];
                                             203  ld r4 [r11+1];
                                             204  mul r5 r1 r3;
                                             205  mul r6 r2 r4;
                                             206  sub r7 r5 r6;
                                             207  st  r7 [r12+0];
                                             208  mul r5 r1 r4;
                                             209  mul r6 r2 r3;
                                             210  add r7 r5 r6;
                                             211  st  r7 [r12+1];
```

Figure 5.4 – **Assembly for Conventional Register Organization.** The address of variable a resides in register r10, the address of b in register r11, and the address of x in register r12. Registers r1–r4 are used to stage the real and imaginary components of a and b. Register r5 and r6 are used to store intermediate results, and register r7 is used to temporarily store the real and imaginary components of the product.

of x resides in r12. Registers r1–r4 are used to stage the real and imaginary components of a and b that are loaded from memory, and registers r5–r7 are used to stage intermediate values.

As shown in Figure 5.4, computing the product of two complex values involves 6 arithmetic operations: 4 scalar multiplications, 1 addition, and 1 subtraction. As shown, the computation also requires 6 memory operations: 4 loads and 2 stores. The computation involves a total of 30 register file operations: 20 reads and 10 writes. The arithmetic operations involved in the computation of the complex product consume approximately 9.8 pJ; the memory operations consume approximately 70.2 pJ; the register file accesses consume approximately 31.1 pJ, which exceeds the energy expended performing the 6 arithmetic operations required by the computation.

```
// source
100  x.re = a.re * b.re - a.im * b.im;
101  x.im = a.re * b.im + a.im * b.re;
```

```
// assembly
200  ld dr0 [ar0+0];
201  ld dr1 [ar0+1];
202  ld dr2 [ar1+0];
203  ld dr3 [ar1+1];
204  mul gr1 dr0 dr2;
205  mul tr0 dr1 dr3;
206  sub tr0 gr1 tr0;
207  st  tr0 [a2+0];
208  mul dr0 dr0 dr3;
209  mul tr0 dr1 dr2;
210  add tr0 dr0 tr0;
211  st  tr0 [ar2+1];
```

Figure 5.5 – Assembly Using Explicit Forwarding and Operand Registers. The address of variable a resides in operand register a0, the address of b in operand register a1, and the address of x in operand register a2. Operand registers d0–d3 are used to stage the real and imaginary components of a and b; operand register d0 is also used to stage the result of the multiplication performed at line 208. Forwarding register t0 is used to explicitly forward results from the ALU.

Figure 5.5 shows the corresponding sequence of assembly instructions using operand registers and explicit forwarding to stage operands and intermediate results. The assembly listing shown in Figure 5.5 assumes that the address of variable a resides in operand register a0, the address of b in operand register a1, and the address of x in operand register a2. Operand registers d0–d3 are used to stage the real and imaginary components of a and b; operand register d0 is also used to stage the result of the multiplication performed at line 208. Forwarding register t0 is used to explicitly forward results from the ALU, which are consumed in both the ALU and XMU.

The assembly listing shown in Figure 5.5 performs the same number of arithmetic and memory operations; consequently, the amount of energy consumed performing the arithmetic operations and accessing memory remains unchanged. However, fewer register file accesses are performed, and most of the register accesses are to operand registers. Explicit forwarding eliminates 10 register file accesses, reducing to 20 the number of register file operations. The number of register file reads is reduced from 20 to 15; only one read operation is performed at the general-purpose register file. The number of register file writes is reduced from 10 to 5; again, only one write operation is performed at the general-purpose register file. As a consequence, the amount of energy spent staging operands in registers decreases from 31.1 pJ to 9.3 pJ. Perhaps significantly, the use of operand registers and explicit forwarding results in less energy being expended staging data in registers than is expended performing the arithmetic operations required by the computation.

5.4 Allocation and Scheduling

Explicit operand forwarding and operand registers require additional compiler passes and optimizations. This section describes the register allocator and instruction scheduler implemented in the compiler used in the

evaluation, and addresses compilation issues associated with allocating registers and scheduling instructions for a processor that provides operand registers and supports explicit operand forwarding.

Scheduling for Explicit Forwarding

A basic explicit operand forwarding compiler pass can be implemented as a peephole optimization performed after instruction scheduling and register allocation. The peephole optimization identifies instruction sequences in which the result produced by an instruction in the sequence can be explicitly forwarded to an instruction that appears later in the sequence and replaces the operand of the later instruction with a forwarding register. In general, the optimization pass cannot replace the destination of the producing instruction with a forwarding register unless it determines that all consumers of the result can receive the result from the forwarding register. Essentially, the optimization pass must know the locations of all uses of the result, and that all uses have been updated to use the forwarding register before modifying the producing instruction. This requires information about the definitions and uses of the result of the producing instruction that is generated by data-flow analysis such as reaching definitions analysis which may not be available during a peephole optimization pass.

In general, an explicit forwarding optimization pass is more effective when performed between instruction scheduling and register allocation. This allows the register allocator to avoid allocating registers to those variables that can be explicitly forwarded. Typically, variables that can be explicitly forwarded are also ideal candidates for operand registers and will displace other variables competing for operand registers; consequently, performing an explicit forwarding pass before register allocation allows more variables to be assigned to operand registers and achieves better operand register utilization.

The optimization pass implemented in the evaluation compiler is performed immediately before register allocation, which occurs after instruction scheduling. The optimization pass identifies instructions whose results are consumed before another instruction executes on the same function unit. If all uses of the variable defined by an instruction appear before another instruction executes on the same function unit, the optimization pass assigns the variable to the forwarding register. Those variables that are assigned to the forwarding registers are ignored during register allocation.

The instruction scheduler can expose opportunities for explicit forwarding by scheduling dependent instructions so that the result of the earlier instruction will be available in the forwarding register when the later instruction reaches the execute stage of the pipeline. To improve locality, the compiler schedules sets of dependent instructions, and attempts to schedule sequences of dependent instructions on the same function unit. This exposes opportunities for the compiler to explicitly forward operands, and to store those that cannot be explicitly forwarded in local operand registers.

Allocating Operand Registers

Operand registers could be allocated using a conventional register allocator provided that it accounts for the restricted connectivity between the operand register files and function units. Any variables that are operands to instructions that execute on different function units must be assigned to general-purpose registers. Variables

that are read by only one function unit could be allocated to the operand registers associated with the function unit. The register allocator could assign each variable to the least expensive register that is available for the variable. However, without considering the broader impact of which variables are assigned to operand registers energy efficiency, we should not expect such a register allocator to select the best variables for operand registers.

The register allocator used in the compiler sorts the variables before assigning registers so that variables that are better candidates for operand registers are assigned registers before variables that are less suitable candidates. After sorting the variables, the compiler scans the sorted list and assigns the least expensive register available when it reaches that variable. The compiler preferentially schedules dependent instructions on the same function unit in an attempt to expose opportunities for using the local operand registers to stage intermediates between dependent instructions.

The register allocation phase proceeds as follows. The register allocator first constructs an interference graph for all of the variables in the compilation unit. The nodes in the interference graph represent variables in the compilation unit, and the edges in the interference graph indicate that the live ranges of the two nodes connected by the edge overlap. Two variables cannot be assigned to the same register if the corresponding nodes in the interference graph are connected by an edge. After constructing the initial interference graph, the register allocator coalesces pairs of nodes that correspond to the source and destination of a copy operation. Both variables represented by a coalesced node will be assigned to the same register, which usually allows one or more move operation to be eliminated.

After coalescing nodes in the interference graph, the register allocator analyzes the compilation unit to determine the order in which variables should be assigned to registers. This phase of the register allocation process produces a sorted list of all of the nodes in the interference graph. Two strategies for ordering the nodes in the sorted list are described in the following sections.

Finally, the register allocator scans the sorted list and assigns registers to the nodes in the interference graph. The allocator assigns the least expensive register available for the node. The allocator uses the interference graph to determine which registers are available. Basically, a register cannot be assigned a register that has already been assigned to an adjacent node in the interference graph. For example, the register allocator will assign a data register to a node representing variables that are only read by the ALU if a data register is available when the allocator encounters the node; otherwise, the allocator will assign a general-purpose register to the node. After selecting a register, the register allocator colors the node in the interference graph with the assigned register.

Should the register allocator encounter a variable that cannot be assigned a register because there are no registers available, it terminates and requests the instruction schedule be modified to spill the variable to memory. The spill instruction, which stores the spilled value to memory, is inserted immediately after the value is produced so that a forwarding register can be used to forward the value, thereby avoiding the need to allocate a register for the spilled variable until it is used. The register allocation process is repeatedly attempted until it completes successfully.

```
// source

100  x.re = a.re * b.re - a.im * b.im;
101  x.im = a.re * b.im + a.im * b.re;

// compiled

200  t1  = load [a.re];   [defs=1 uses=2 appearances=3]
201  t2  = load [a.im];   [defs=1 uses=2 appearances=3]
202  t3  = load [b.re];   [defs=1 uses=2 appearances=3]
203  t4  = load [b.im];   [defs=1 uses=2 appearances=3]
204  t5  = mult t1 * t3;  [defs=1 uses=1 appearances=2]
205  t6  = mult t2 * t4;  [defs=1 uses=1 appearances=2]
206  t7  = sub  t5 - t6;  [defs=1 uses=1 appearances=2]
207  [x.re] = store t7;
208  t8  = mult t1 * t4;  [defs=1 uses=1 appearances=2]
209  t9  = mult t2 * t3;  [defs=1 uses=1 appearances=2]
210  t10 = add t8 + t9;   [defs=1 uses=1 appearances=2]
211  [x.im] = store t10;

// data-vars    = [t1 t2 t3 t4 t5 t6 t7 t8 t9 t10]
// address-vars = [a b x ]
```

Figure 5.6 – **Calculation of Operand Appearances.** The code computes the product of two complex values. The order in which variables are assigned registers is listed at the bottom. Variables `t6`, `t7`, `t9`, and `t10` could be explicitly forwarded.

Operand Appearance

One metric for ordering the nodes in the interference graph is the number of times the variable will be accessed when the code executes. In general, it is not possible to determine this, particularly not as a static compiler analysis. However, we can estimate the number of times a variable will be accessed by counting the number of times it appears in the compilation unit. Presumably, variables that appear more often will be accessed more often when the code executes, and it will be profitable to assign these frequently occurring variables to inexpensive operand registers.

Of course, simply counting the number of appearances of a variable does not account for the structure of the control-flow in the compilation unit. For example, variables that are accessed within deeply nested loops are likely to be accessed more often that variables that are accessed outside of loops. A reasonable way to account for this is to weight each appearance of a variable, weighting appearances that are deeper in a loop nest more heavily. The *operand appearance* metric accounts for any loops in the compilation unit by weighting each definition and use of a variable by the loop depth at which the use occurs.

Operand Intensity

The operand appearance metric identifies variables that are likely to be accessed more often when the code executes and prioritizes variables that are accessed more often. However, it fails to account for the lifetimes of the variables, which affects how long individual variables will occupy registers. Consider allocating registers

```
// source

100  x.re = a.re * b.re - a.im * b.im;
101  x.im = a.re * b.im + a.im * b.re;

// compiled

200  t1  = load [a.re];   [defs=1 uses=2 life=8 intensity=0.38]
201  t2  = load [a.im];   [defs=1 uses=2 life=8 intensity=0.38]
202  t3  = load [b.re];   [defs=1 uses=2 life=7 intensity=0.43]
203  t4  = load [b.im];   [defs=1 uses=2 life=5 intensity=0.60]
204  t5  = mult t1 * t3;  [defs=1 uses=1 life=2 intensity=1.00]
205  t6  = mult t2 * t4;  [defs=1 uses=1 life=1 intensity=2.00]
206  t7  = sub  t5 - t6;  [defs=1 uses=1 life=1 intensity=2.00]
207  [x.re] = store t7;
208  t8  = mult t1 * t4;  [defs=1 uses=1 life=2 intensity=1.00]
209  t9  = mult t2 * t3;  [defs=1 uses=1 life=1 intensity=2.00]
210  t10 = add t8 + t9;   [defs=1 uses=1 life=1 intensity=2.00]
211  [x.im] = store t10;

// data-vars    = [t10 t9 t7 t6 t8 t5 t4 t3 t2 t1]
// address-vars = [a b x ]
```

Figure 5.7 – **Calculation of Operand Intensity.** The code computes the product of two complex values. The order in which variables are assigned registers is listed at the bottom. Data register candidates `t10`, `t9`, `t7`, and `t6`, which are prioritized because they have the greatest operand intensity, could be explicitly forwarded

for variables in the code shown in Figure 5.6, which has been annotated with the operand appearances metric computed for each variable where it is defined. Let us assume that the target machine has a single operand register and does not support explicit forwarding. A register allocator using the operand appearance metric would allocate one of `t1` – `t4` to the operand register, as these variables have the greatest operand appearance values. To make the example concrete, let us assume that `t1` is assigned to the operand register and occupies the register from line 201 through 208. The operand register is used 3 times between 201 and 208. The operand register would have been more productively employed if instead the register allocator had assigned the operand register to variables `t5`, `t7`, and `t8`, as it would be used 6 times between line 201 and 208.

Intuitively, the operand bandwidth captured in operand registers can be increased by allocating operand registers to variables that are either short-lived or are long-lived and frequently accessed. The intuition here is that short-lived variables will not occupy an operand register for very long, so it will be available for use by other variables. The register allocator can identify variables that exhibit these properties by computing a measure of *operand intensity* for each variable. Operand intensity is computed by dividing the operand appearance value for a variable by its estimated lifetime. The lifetime is estimated statically as the number of instructions for which the variable is live, as computed by a conventional live-variables data-flow analysis. The lifetime estimate does not account for the loop structure. This is intentional. The primary purpose of including the estimated lifetimes of variables is used to order appearances at the same level of loop nesting. An example of the computation appears in Figure 5.7.

5.5 Evaluation

Figure 5.8 shows the normalized operand bandwidth at the register files and memory interface for several embedded kernels. The figure shows the composition of the operand read bandwidth and the result write bandwidth as additional operand registers are made available to the compiler. Pairs of adjacent columns illustrate the impact of explicit forwarding: the left column shows bandwidth composition with explicit forwarding, and the right column shows the composition without explicit forwarding. The dedicated accumulator registers, which are used by multiply-and-accumulate operations, are integrated with the ALU and are reported as operand registers. The accumulators appear in all of the configurations, and contribute to the operand register bandwidth reported for the configuration that lacks operand register files. The configuration with 8 operand registers, which associates 4 operand registers with the ALU and 4 with the XMU, corresponds to the Elm register organization. To illustrate the impact of operand register organization, the read and write bandwidths are normalized to a conventional register organization with a unified register file with 4 read ports and 2 write ports. The data shown in the figures were collected using a general-purpose register file with 2 read ports and 2 write ports. The reduction in the read port bandwidth available at the general-purpose register file can result in additional register accesses, as some shared data may be duplicated; it also may increase kernel execution times, as additional cycles may be required to resolve read port conflicts. The data is normalized to correct for additional register accesses.

The number of general-purpose registers remains constant at 32 register for all configurations. The additional operand registers increase the aggregate number of registers that are available to the compiler, which allows the compiler to reduce the number of variables that are spilled to memory. Both the **dct** and **huffman** kernels show a reduction in aggregate operand bandwidth because there are fewer register spills, which introduce load and store operations, both of which read registers. The **jpeg** kernel shows a reduction for the same reason, and includes code similar to the **dct** and **huffman** kernels. However, the aggregate benefit that can be attributed simply to the additional registers is modest, as the compiler introduces few register spills even when there are no operand registers.

Figure 5.8 shows that operand registers are an effective organization for capturing a significant fraction of operand bandwidth local to the function units. Without explicit forwarding, the configuration with 8 operand registers, which provides four operand registers per function unit, captures between 32% and 66% of all operand bandwidth in operand registers; the operand registers account for 43% to 87% of all register references. As the figure illustrates, explicit operand forwarding replaces operand register references with explicit operand forwards. With explicit operand forwarding, the configuration with 8 operand registers captures between 25% and 64% of all operand bandwidth in operand registers; the operand registers account for 38% to 81% of all register references.

In addition to capturing instruction-level producer-consumer data locality within the function units, explicit operand forwarding reduces register pressure. The **crc** and **crosscor** kernels clearly illustrate the effectiveness of explicit operand forwarding at reducing operand register pressure when the number of operand registers is limited; the configurations that use explicit operand forwarding are able to capture a greater fraction of data references in operand registers.

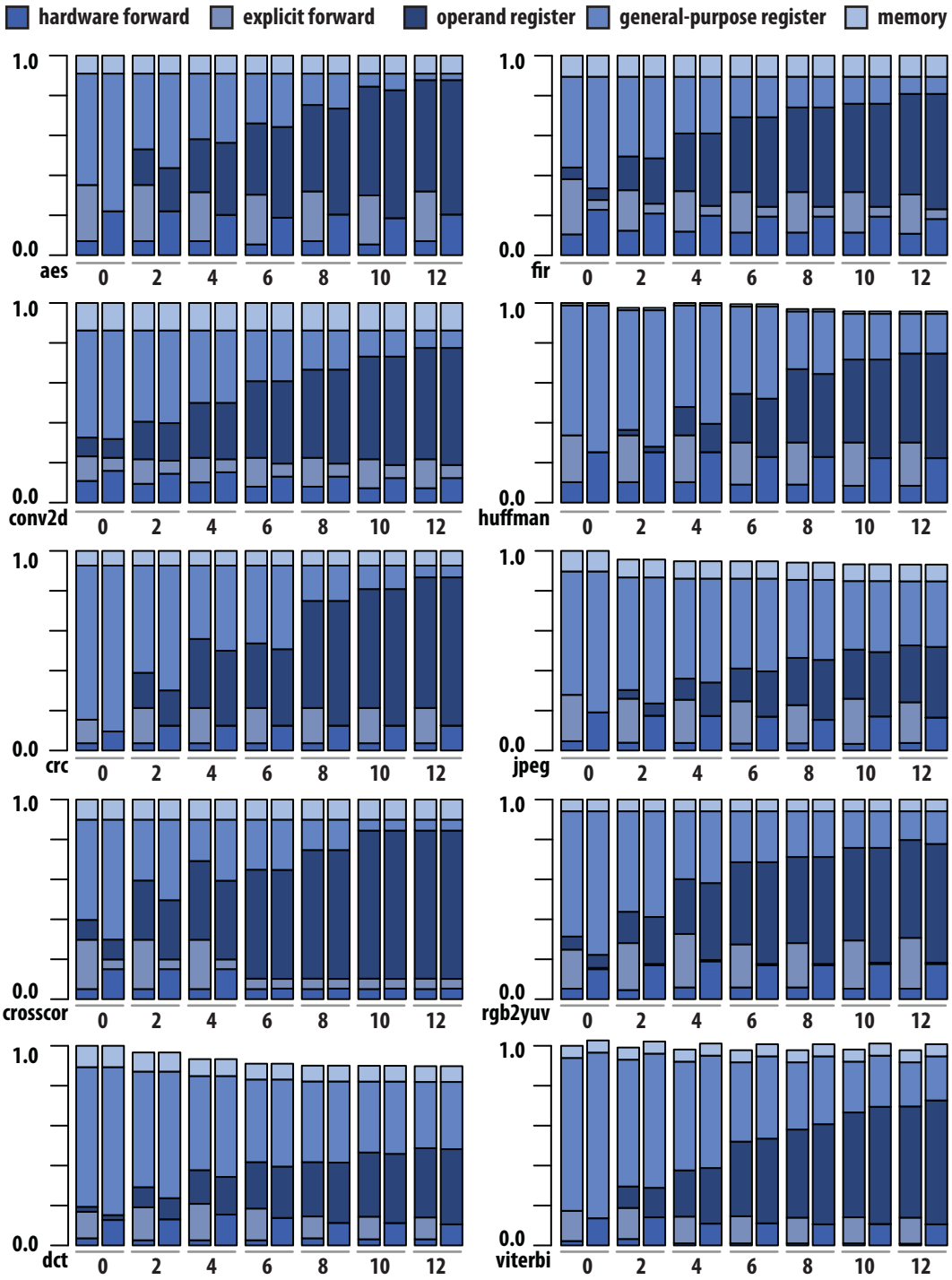


Figure 5.8 – **Impact of Operand Registers on Data Bandwidth.** The number of operand registers appears on the horizontal axis. Pairs of adjacent columns illustrate the impact of explicit forwarding: the left column shows bandwidth composition with explicit forwarding, and the right column shows the composition without explicit forwarding.

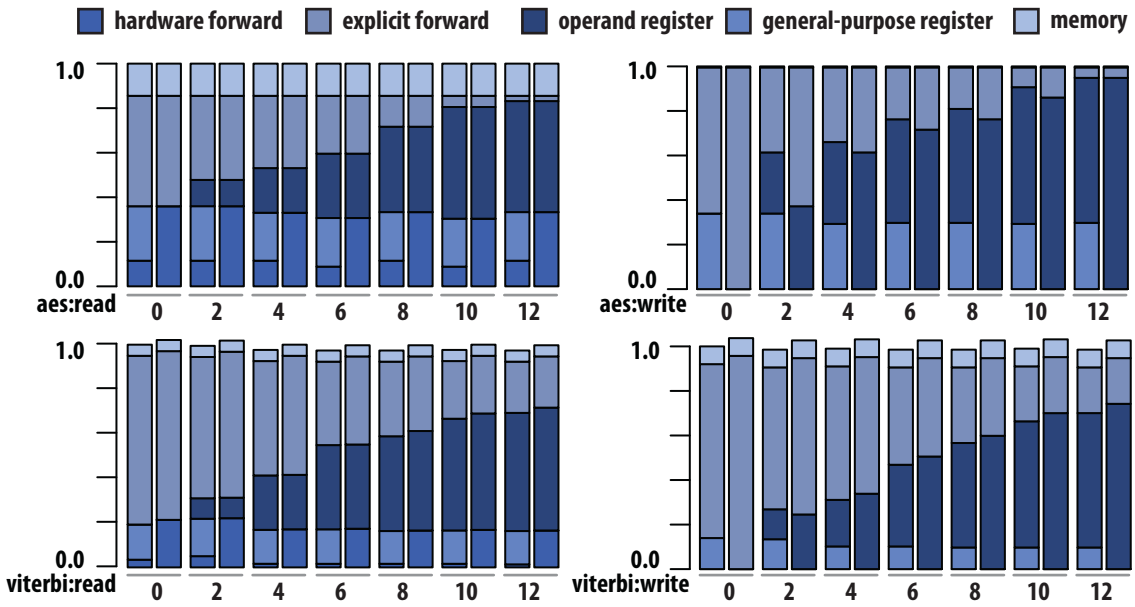


Figure 5.9 – **Operand and Result Bandwidth.** The number of operand registers appears on the horizontal axis. Pairs of adjacent columns illustrate the impact of explicit operand forwarding on the operand and result bandwidth: the left column shows the bandwidth composition with explicit forwarding; the right column shows the bandwidth composition without. The data shown are normalized to illustrate the impact clearly; the normalization accounts for differences in the number of operand reads and result writes. The result write bandwidth data are shown separately.

Perhaps more interesting, the **viterbi** kernel data shows that explicit operand forwarding can compensate for the reduction in the number of read ports at the general purpose register file. Explicit operand forwarding effectively provides additional named locations that the compiler uses to deliver operands to instructions, which compensates for the reduction in read ports when instruction sequences exhibit sufficient instruction-level produce-consumer data locality. It is possible for conventional hardware forwarding to provide the same effective increase in operand bandwidth when the control logic in the forwarding network uses the register identifiers encoded in instructions rather than the addresses that are sent to the register files to detect hazards. This requires arbitration hardware at the register file to select those register accesses that cannot be forwarded, and the compiler must be designed so that the register allocator exploits the ability of the forwarding network to deliver the operands despite apparent conflicts. The register allocator implemented in the elmcc compiler considers register file port conflicts without evaluating whether operands might be forwarded, which accounts for the behavior observed in the figure.

Figure 5.9 shows the impact of explicit forwarding on the composition of the operand and result bandwidth for the **aes** and **viterbi** kernels. As the figure illustrates, explicit forwarding does not effect the aggregate **aes** bandwidth. Explicit forwarding allows from 30% to 33% of all results to be forwarded without writing a register file, and allows from 22% to 24% of all operands to be explicitly forwarded. The difference in the number of results and operands that are explicitly forwarded simply reflects the average number

of operands to an instruction exceeding the average number of result values. Explicit forwarding reduces operand register pressure, and allows more data bandwidth to be captured close to the function units. This is apparent from the impact explicit operand forwarding has on the **aes** write bandwidth, which shows that explicit forwarding directs more of the result bandwidth to registers that are integrated with the function units. The number of register file reads is not affected because the pipeline control logic detects that an operand will be forwarded and suppresses unnecessary register file accesses.

The **viterbi** kernel exhibits a reduction in aggregate data bandwidth when the compiler uses explicit operand forwarding. The reduction results from the compiler eliminating operand and general-purpose register accesses that communicate operands across multiple instruction pairs. Essentially, the compiler uses the temporary registers to explicitly communicate data between instructions that execute multiple cycles apart. When ephemeral data is explicitly forwarded, a register write and subsequent read are eliminated, which reduces the aggregate data bandwidth and keeps more of the data bandwidth local to the function units.

Energy Efficiency

Figure 5.10 shows the energy expended delivering data to function units. To illustrate the impact of the register organization and explicit operand forwarding, the data has been normalized within each kernel. The configuration with no operand registers uses a general-purpose register file with 4 read ports; the other configurations use a general-purpose register file with 2 read ports. Because the register file is physically smaller and the internal cell capacitances are reduced, less energy is expended accessing the register file with 2 read ports. However, to avoid conflating the reduction in the general-purpose register file access energy, we have used the 4 port register file access energy for all of the configurations. We have also assumed that the local memory that backs the register files is a 2 kB direct-mapped cache rather than the software-managed Ensemble memory used in Elm. This reflects a more conventional processor organization, but makes memory accesses more expensive. Consequently, the data presented in the figure provides a conservative estimate of the energy efficiency improvements achieved from the operand register organization.

As the figure clearly illustrates, operand registers and explicit operand forwarding both reduce the energy expended delivering data to function units. Operand registers alone reduce the energy consumed staging data transfers to function units by 19% to 32% for configuration with 4 operand registers per function unit; they reduce the energy consumed in the registers files and forwarding network by 31% to 56%. Combined, operand registers and explicit forwarding reduce the energy consumed staging data transfers to function units by 19% to 38% for the configuration with 4 operand registers per function unit; they reduce the energy consumed in the registers files and forwarding network by 35% to 57%.

Operand registers effectively filter accesses to more expensive registers. Increasing the number of operand registers allows the operand registers to filter more accesses. As Figure 5.10 shows, this improves energy efficiency by reducing the energy consumed accessing the general-purpose register file. As we provision additional operand registers, the amount of energy expended accessing operand registers increases. In part, this is because operand registers are accessed more often. Additionally, the larger operand register files are more expensive to access, and the efficiency advantage offered by small register files diminishes as we

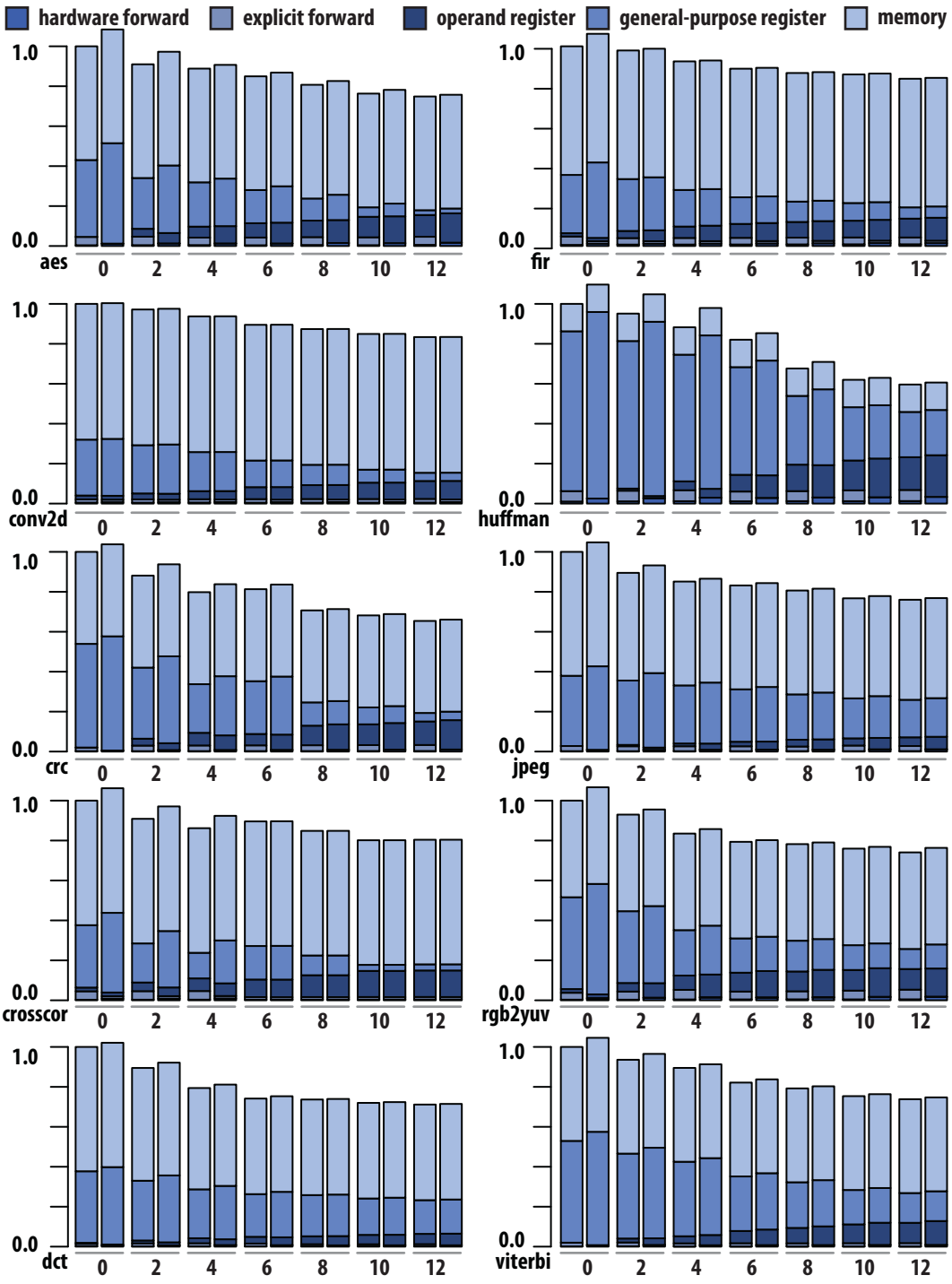


Figure 5.10 – **Impact of Register Organization on Data Energy.** The number of operand registers appears below each column.

introduce additional operand registers. To compensate, one could segment the bit-lines within the operand register files to allow some operand registers to be accessed less expensively. However, we have found that the capacity of the operand register files is typically limited by area efficiency considerations rather than energy efficiency considerations: operand register files must be small enough to provide sufficiently fast access times to avoid introducing critical paths that will limit the frequency at which the processor will operate at.

As should be evident from contrasting the data presented for the different kernels, data working sets have a significant impact on the effectiveness of operand registers and explicit forwarding. Consider the breakdown of operand bandwidth for the **aes** and **rgb2yuv** kernels shown in Figure 5.8, and what we can infer about the working sets of the two kernels from the data. The slow increase in operand register bandwidth in **aes** as the number of operand registers increases reflects the lack of small working sets with significant short-term locality and reuse for the compiler to promote to operand registers. Conversely, the significant fraction of the operand bandwidth in **rgb2yuv** captured with few operand registers reflects the compiler promoting a critical set of high-intensity variables to operand registers. Four data and four address registers are sufficient to capture the critical working set of **rgb2yuv**, and further increasing the number of operand registers results in little additional bandwidth capture, as only low-intensity variables are promoted to operand registers. Despite contributing a small fraction of all data references, memory accesses are considerably more expensive than register file accesses and are responsible for the preponderance of the data access energy.

Compilers and Explicit Operand Forwarding

With operand registers and explicit operand forwarding providing detailed control over the movement of data throughout the pipeline, it is reasonable to ponder whether we could rely exclusively on the compiler to forward all operands explicitly. This would eliminate the need for dedicated hardware to detect dependencies between instructions within the processor pipeline. The significant disadvantage of removing the hardware that detects dependencies and automatically forwards operands is that it tends to increase the number of instructions comprising kernels. The compiler may need to schedule explicit idle cycles to ensure the results have reached their destination register before a dependent instruction reads the register. We have found this to be particularly problematic near points where control-flow converges, as the compiler must conservatively schedule instructions to ensure that all dependencies are satisfied regardless of where control arrives from. Another issue is that the compiler must use operand registers to transfer results to dependent instructions when a region lacks sufficient instruction-level parallelism to hide both pipeline and function unit operation latencies; variables that are assigned to operand registers for this reason may otherwise be inferior candidates.

Figure 5.11 shows the impact of relying exclusively on the compiler to forward operands. Pairs of adjacent columns show the benefit of hardware detecting read-after-write hazards and managing the forwarding network: the left column shows the execution time when the pipeline control logic detects hazards and forwards operands as necessary; the right column shows the execution time when the compiler must detect hazards and explicitly forward operands when possible. The number of operands registers appears below each column. The operand registers are distributed evenly between the ALU and XMU. All of the configurations have a general-purpose register file with 2 read ports and 2 write ports.

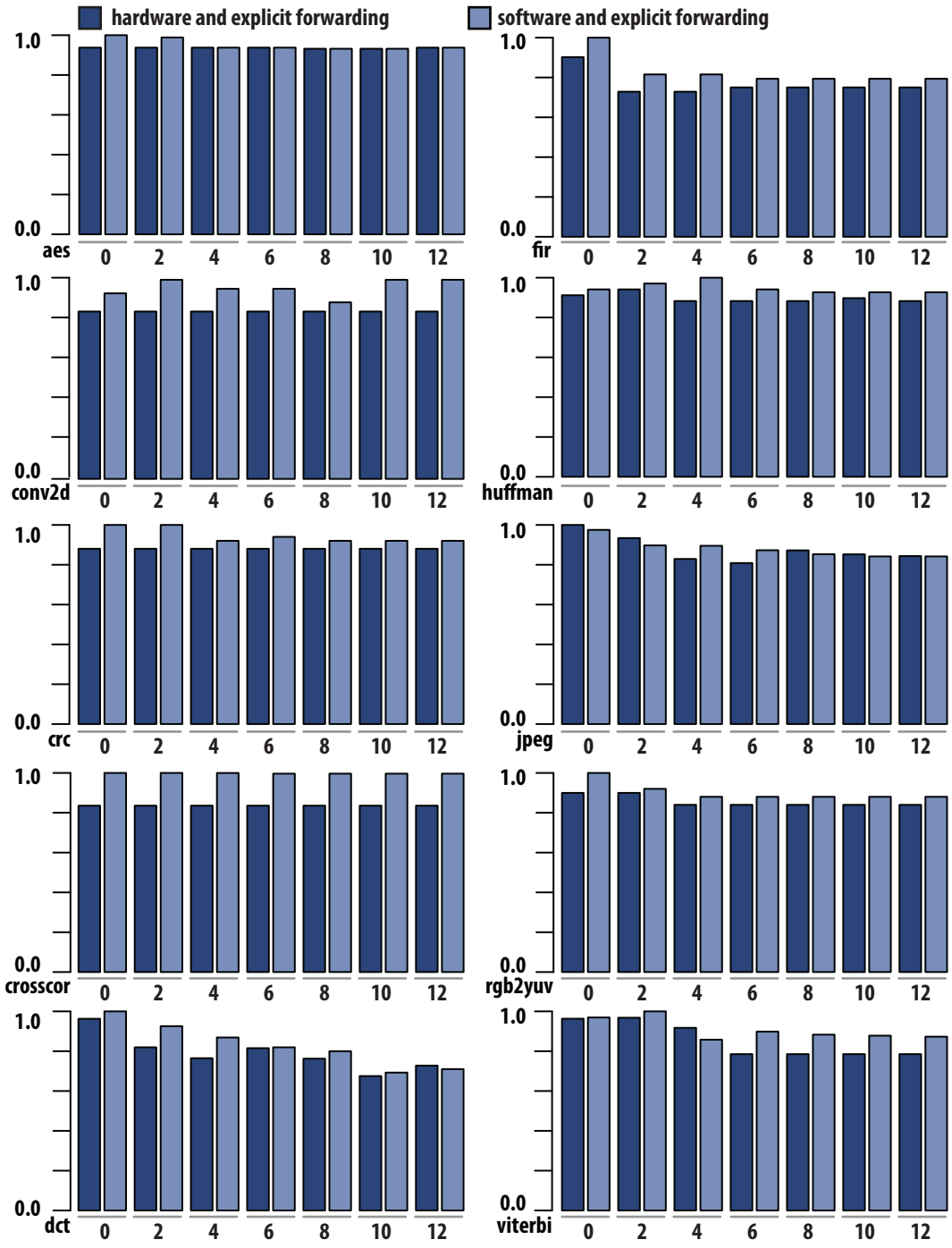


Figure 5.11 – **Execution Time.** Pairs of adjacent columns show the benefit of hardware operand forwarding: the left column shows the execution time with hardware forwarding; the right column shows the execution time when the compiler must detect hazards and explicitly forward operands. The number of operand registers appears below.

Because the execution time reported is for compiled kernels, the execution times exhibit some artifacts that are introduced by the heuristics used in the register allocation and instruction scheduling phases of the compiler. These artifacts appear as an increase in the number of operand registers producing a small increase in the execution time. Typically, this results when the compiler assigns a variable that is an operand to instructions that execute in different function units to an operand register and is unable to schedule a copy operation without introducing an additional cycle to the instruction schedule.

Increasing the number of operand registers allows the operand register files to capture more data references, which reduces the demand for operand bandwidth at the general-purpose register file. The reduction in the execution times we observe when the number of operand registers increases results primarily from a reduction in the number of port conflicts at the general-purpose register file. The notable exception is the **dct** kernel. The additional register capacity allows more of the data working set to be captured in registers, which reduces the number of the memory operations that are executed. The compiler used to compile the kernels does not exploit the ability of the forwarding network to deliver more than 2 operands in the decode stage, which effectively allows more than 2 operands to be provided at the general-purpose register file when some are forwarded. Instead, the compiler conservatively assumes that an instruction pair may contain at most 2 operands that reside in general-purpose registers. This results a perceptible performance degradation for the 8 operand register configuration of the **dct** kernel.

As Figure 5.11 shows, the performance of kernels may degrade significantly when we rely exclusively on the compiler to forward operands. For example, the throughput of the **crosscor** kernel decreases by 16%. The energy expended in the instruction register files exceeds the modest savings realized by eliminating the hardware that is implemented to detect dependencies. Other kernels, notably the **aes** and **crc** kernels, perform reasonably well when we rely exclusively on the compiler provided that we provision a sufficient number of operand registers to capture the short-term instruction-level dependencies within the performance-critical regions of the code.

Technology and Design Considerations

Explicit forwarding and operand registers provide greater benefits when VLSI technology and design constraints limit the efficiency of multi-ported register files and memories. For example, often memory compilers are optimized for large memories and are not designed to generate efficient small memories. Without access to efficient small memories and register files, the cost of staging operands in general-purpose registers increases, which improves the relative efficiencies provided by operand registers and explicit forwarding. Similarly, standard cell libraries rarely allow wide multiplexers to be implemented as efficiently as is possible when a datapath is constructed using custom design and layout techniques, which may increase the cost of implementing the multiplexers that are needed to read operand registers, making explicit forwarding comparatively more attractive.

5.6 Related Work

Distributed [122], clustered [40], banked [30], and hierarchical [154] register organizations improve access times and densities by partitioning registers across multiple register files and reducing the number of read and write ports to each register file. The number of ports needed to deliver a certain operand bandwidth can be reduced by using SIM register organizations [116] to allow each port to deliver or receive multiple operands to multiple function units. Such SIMD register organizations are suitable for the subset of data parallel applications that are readily vectorized, but impose inefficiencies when data and control dependencies prevent data parallel computations from being vectorized.

The hierarchical CRAY-1 register organization [123] associates dedicated scalar and address registers with groups of scalar and address function units and provides dedicated backing register files to filter spills to memory. In contrast, operand register files are significantly smaller, optimized for access energy, and each is integrated with a single function unit in the execute stage of the pipeline to reduce the cost of transferring operands between an operand register file and its dedicated function unit. This allows operand registers to achieve greater reductions in energy consumption in processors with short operation latencies and workloads with critical working sets and short-term producer-consumer locality that can be captured in a small number of registers. Operand registers further differ by allowing both levels of the register hierarchy to be directly accessed. This avoids the overhead of executing instructions to explicitly move data between the backing register files and those that deliver operands to the function units [154]. It also avoids the overhead of replicating data in multiple register files [30].

Horizontally microcoded machines such as the FPS-164 [142] required that software explicitly route all results and operands between function units and register files. This work introduces opportunistic explicit software forwarding into a conventional pipeline, where automatic forwarding performed by the forwarding control logic avoids the increase in code size and dynamic instruction counts needed to explicitly route all data.

Hampton described a processor architecture that exposes pipeline registers to software[57] and allows the result of an instruction to be directed to a pipeline register. Like explicit bypass registers, this exposed architecture allowed software to replace some writes to a register file with writes to pipeline registers. Hampton reports similar data on the number of register file writes that can be eliminated. The architecture is limited in that a result could be written to at most one pipeline register, so only operands that are accessed by a single instruction could be communicated explicitly through the exposed pipeline registers. Operands with multiple consumers must be assigned to general purpose registers. The architecture relied on software restart markers to recover from exceptions that might alter the state of the pipeline registers [58].

5.7 Chapter Summary

Operand register files are small, inexpensive register files that are integrated in the execute stage of the pipeline; explicit operand forwarding lets software opportunistically orchestrate the routing of operands through the forwarding network to avoid writing ephemeral values to registers. Both operand registers and

explicit operand forwarding let software capture short-term reuse and locality in inexpensive registers that are integrated with the function units. This keeps a significant fraction of operand bandwidth local to the function units, and reduces the energy consumed communicating data through registers and the forwarding networks.

Chapter 6

Indexed and Address-Stream Registers

This chapter introduces the concepts of indexed registers and address-stream registers, and describe an architecture that uses indexed registers and address-stream registers to expose hardware for performing vector memory operations to software.

6.1 Concepts

Registers are used to store intermediate values and to stage data transfers between function units and memory. In an architecture such as Elm that provides operand registers, general-purpose registers are used to capture working sets that exhibit medium-term reuse and locality. Providing additional general-purpose registers allows software to capture larger working sets in registers, reducing the amount of data that must be transferred between registers and memory. This improves energy efficiency and performance, as registers are less expensive and faster to access than memory. However, exposing additional architectural registers to software requires the use of instruction encodings that dedicate more bits for encoding register names. This leaves fewer bits available for encoding operations or increases the number of bits required to encode an instruction, neither of which are desirable.

Perhaps more significantly, many compute-intensive kernels, and particularly those with enough structured reuse to benefit from large register files, access data in ways that makes it difficult to exploit large register files without using software techniques, such as loop unrolling, to expose additional scalar variables that can be assigned to registers. A kernel that operates on the elements of an array serves to illustrate this point. Before assigning the elements of the array to registers, the compiler must typically replace each reference to an element of the array with an equivalent reference to a scalar variable that is introduced explicitly for this purpose, a transformation commonly referred to as scalar replacement. If a kernel accesses the array in a loop using an index that is a non-trivial affine function of the loop induction variable, the loop must be unrolled to enable the scalar replacement transformation. Because loop unrolling replicates operations within the bodies of loops, loop unrolling increases the number of instructions in the critical instruction working sets and kernels of an application. Reductions in the energy consumed staging operands in registers may be offset

by increases in the energy consumed loading instructions from memory. Consequently, software techniques such as loop unrolling that exploit additional registers but introduce additional instructions are unattractive for architectures such as Elm because the additional instructions increase instruction register pressure.

Before continuing, we should observe that the aggregate number of general-purpose registers that may be accessed by the instructions residing in the instruction registers will be limited to a small multiple of the number of instruction registers. This follows from each instruction being able to reference at most 3 general-purpose registers, though many instructions will reference fewer. There are techniques for allowing software to access a larger number of physical registers. For example, vector processors are able to provide a large number of physical registers and offer compact instruction encodings by associating a large number of physical registers with a single vector register identifier. Similarly, architectures that provide register windows allow software to access different sets of physical registers by rotating through distinct windows, though these are typically used to avoid saving and restoring registers at function call boundaries, and when entering and terminating interrupt and trap handlers [114].

Index registers let software access some subset of the architectural registers indirectly. The registers that can be accessed indirectly through the index registers are referred to as indexed registers. Some of the indexed registers may be architectural registers that can be accessed using a conventional register name; others may be dedicated indexed registers that can only be accessed through index registers. Index and indexed registers improve efficiency by allowing larger data working sets to be efficiently stored in the register hierarchy; they let software increase the fraction of intra-kernel data locality that can be compactly captured in registers without resorting to software techniques such as loop unrolling that increase the instruction working set of a kernel. Thus, index and indexed registers simultaneously address the problem of allowing a processor to increase the number of registers that are available to software without requiring instruction set changes and the problem of allowing software to exploit addition registers without resorting to techniques such as loop unrolling that increases the number of instructions in the working sets of the important kernels within embedded applications. Like vector registers, index and indexed establish an association between a large number of physical registers and a register identifier that is exposed to software. However, index and indexed are more general and more flexible: index and indexed registers expose the association to software and let software control the assignment of physical registers to register identifiers, and allow the physical registers to be accessed using multiple register identifiers.

Vector memory operations transfer multiple data words between memory and the indexed registers, which improves code density by allowing a single vector memory instruction to encode multiple memory operations. Techniques that improve code density improve efficiency in general, and are particularly attractive for architectures such as Elm that achieve efficiency in part by delivering a significant fraction of instruction bandwidth from a small set of inexpensive instruction registers. Vector memory operations provide a means for software to explicitly encode spatial locality within the operation, which allows hardware to improve the scheduling and handling of the memory operations. Decoupling allows the memory operations that transfer the elements of a vector to complete while the processor continues to execute instructions, which lets software schedule computation in parallel with memory operations. Decoupling vector memory operations also assists software in tolerating memory access latencies, as vector memory operations can be initiated early to

allow the operations more time to complete.

The Elm architecture exposes the hardware that generates sequences of addresses for vector memory operations as address-stream registers. The address-stream registers deliver sequences of addresses and offsets that can be combined with other registers to compute the effective addresses of memory operations. Software controls the address sequences computed by the address-stream registers by loading configuration words from memory. Software can save the state of an address sequence by storing the address-stream register to memory, and can later restore the sequence by loading the register from memory. This lets software map multiple address sequences onto the address-stream registers. Software can move elements from the address sequences to other registers, which allows software to implement complex address calculations using a combination of address-stream registers and general-purpose and address registers.

Exposing the address generators as address-stream registers allows software to use the hardware to accelerate repetitive address calculations, improving both performance and efficiency by eliminating instructions that perform explicit address computations from the instruction stream. It also simplifies the task of decomposing a large transfer that can be compactly expressed using vector memory operations and address-stream registers into a sequence of smaller operations that better exploit the storage available in the register organization: the address-stream registers maintain the state of the operation between transfers.

6.2 Microarchitecture

Figure 6.1 illustrates where index, indexed, and address-stream registers reside in the register organization. The index and indexed registers reside with the general-purpose registers and are read early in the pipeline. The address-stream registers are located with the XMU in the execute stage of the pipeline, where they effectively extend the address register file. The following sections describe microarchitectural considerations and the implementation of indexed and address-stream registers in the Elm.

Index and Indexed Registers

The instruction set architecture allows the index registers to be implemented as an extension of the general-purpose registers. The Elm instruction set architecture allows a processor to implement between 32 and 2048 indexed registers, and defines indexed registers **`xr0`** though **`xr31`** to be aliases of general-purpose registers **`gr0`** through **`gr31`**. Consequently, index registers **`xr0`** through **`xr31`** may use the same physical memory as is used to implement the general-purpose registers. Any additional indexed registers that a processor implements require dedicated memory.

When the number of indexed registers exceeds the number of general-purpose registers by a small factor, the indexed registers and general-purpose registers can be implemented efficiently as a single unified register file. Though any additional capacity required for the indexed registers increases the cost of accessing general-purpose registers, a unified register organization improves area efficiency by avoiding the duplication of many of the peripheral circuits found in a register file.

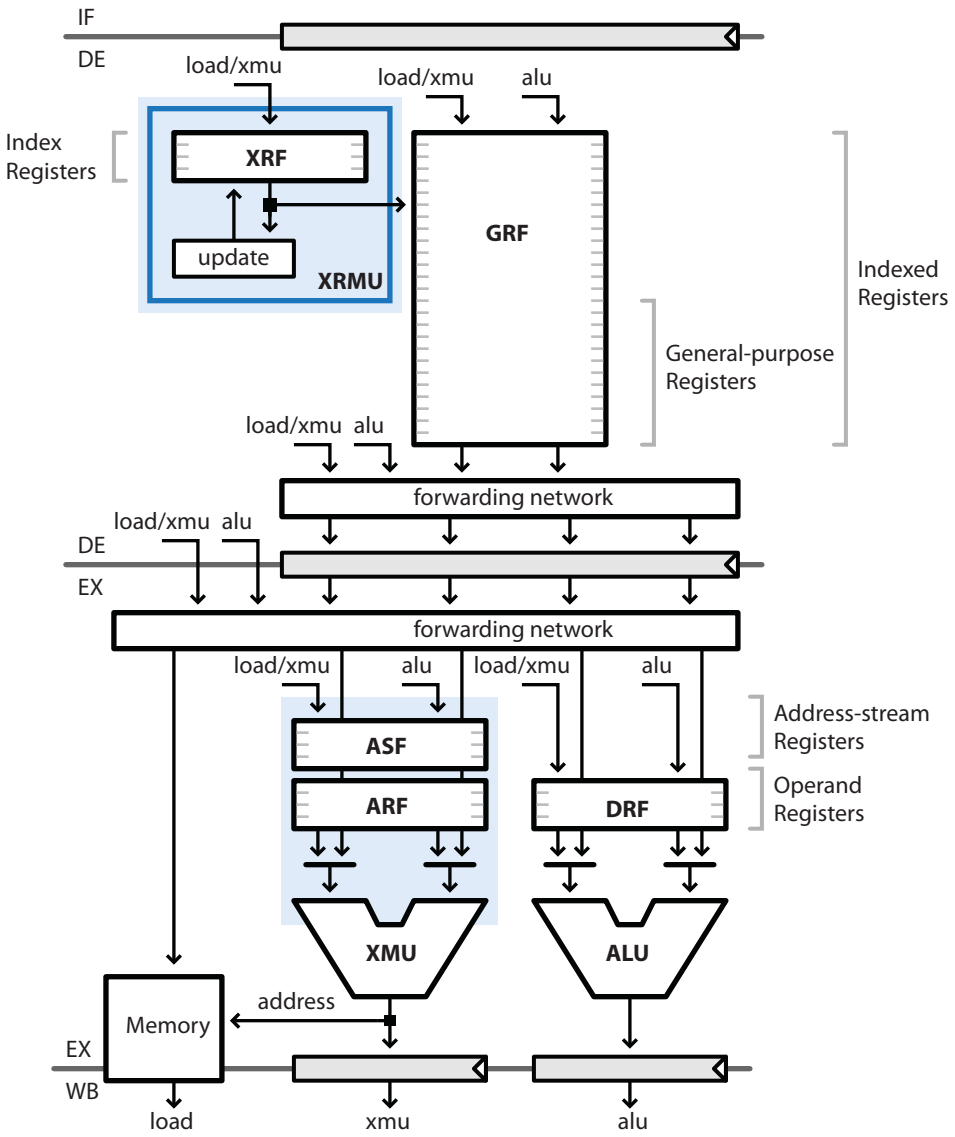


Figure 6.1 – **Index Registers and Address-stream Registers.** Index registers allow registers in the general-purpose register file to be accessed indirectly through the index registers. The index registers reside in the index register file (XRF). The index register management unit (XRMU) implements logic for automatically updating pointers stored in the index registers after each reference. Address-stream registers extend the address registers integrated with the XMU, and are used to automate the generation of fixed sequences of addresses. Hardware associated with the address-stream registers automatically updates the value stored in an address-stream register after each reference. The address-stream registers are implemented in the address-stream register file (ASF).

When the number of indexed registers greatly exceeds the number of general-purpose registers, the additional physical registers may be implemented as a dedicated register file to avoid increasing the cost of accessing the general-purpose registers. This register organization offers the advantage of provisioning addition register file ports. A unified register organization limits the number of operands that can be delivered from the general-purpose registers and indexed registers. The partitioned register file organization allows 2 operands to be delivered from general-purpose registers and 2 operands to be delivered from indexed registers. Perhaps more significantly, it provides an additional 2 write ports, allowing data from the memory interface to be returned to the indexed registers while the XMU writes results back to the general-purpose registers.

Forwarding

Elm automatically forwards values that are stored to indexed registers. Because an index register and a general-purpose register identifier may refer to the same physical register, index register identifiers cannot be used to detect when it is necessary to forward a result. Essentially, the possibility of aliases, introduced by indirection, prevents the pipeline control logic from inspecting the index register identifiers encoded in instructions to detect data dependencies. Fortunately, this alias problem is easily solved. The key insight in forwarding indexed registers is that the physical addresses of the indexed register, not the index register identifiers encoded in instructions, should be used to detect dependencies. Accordingly, the pipeline control logic implemented in Elm uses the physical addresses of any indexed registers to detect when data should be forwarded. Similarly, the pipeline interlocks also use physical register addresses to detect data hazards that the forwarding network cannot resolve. Even if different register files were used for general-purpose registers and indexed registers, it would still be necessary for the pipeline control logic to use physical register addresses to detect dependencies because the sets of indexed registers accessed through different index registers may not be disjoint, and consequently multiple index registers may refer to the same indexed register.

Source indexed register addresses are determined when the index registers are accessed in the decode stage, and advance along with operands read from the indexed register through the pipeline. Destination indexed register addresses are determined in the execute stage of the pipeline. The destination indexed register addresses are used to forward results that are destined for indexed registers when the results are sent to the indexed register file in the write-back stage. Though the destination addresses could be determined in the write-back stage, the time required to access the index registers would contribute to the delay associated with forwarding operands through the forwarding network, and would adversely impact the frequency at which the processor could operate. Once the addresses of the destination and source indexed registers are available, the forwarding hardware simply forwards indexed registers as though they were general-purpose registers, using the indexed register addresses to determine where results should be sent through the forwarding network.

Interlocks

The processor must ensure that an operation that modifies an index register appears to complete before subsequent operations that use the index register to access the indexed registers are initiated. The forwarding

hardware ensures that operations that implicitly modify the index registers appear to complete in the order that software expects. However, it does not ensure that operations that explicitly modify the index registers appear to complete as expected. For example, the **ld.rest** operation in the following assembly fragment must complete and write index register **xp0** before the **add** instruction uses **xp0** to read its first operand from the indexed registers.

```
nop, ld.rest xp0 [ar0+16];
add xp0 xp0 dr1, nop;
```

Because operations that explicitly modify the index register are infrequent, the processor handles hazards associated with software explicitly modifying an index register by stalling the pipeline. The interlock control hardware detects when an instruction in the decode stage attempts to read an operand from the indexed registers using an index register that has a load pending and prevents the instruction from departing the decode stage until the index register has been written.

Address-Stream Registers

The address-stream registers extend the address registers. As an extension of the address registers, address-stream registers are accessed in the execute (EX) stage of the pipeline.

The Elm instruction set architecture requires that updates of address-stream registers appear to be performed in parallel with the execution of the XMU operation that receives its operands from them. It is uncommon for an XMU operation to receive multiple operands from address-stream registers. However, the hardware required to route operands from address-stream registers to the arithmetic units that are responsible for updating the registers after each reference is greatly simplified if each read port of the address-stream register file has a dedicated arithmetic unit. Because the simple arithmetic operations required to update an address-stream register are inexpensive in contemporary VLSI technologies, Elm assigns a dedicated update unit to each of the address-stream register file read ports. This allows the XMU to complete 3 arithmetic operations in parallel. For example, when the following instruction pair executes, the XMU adds the contents of address-stream registers **as0** and **as1** and updates both address-stream registers in parallel.

```
nop,  add dr0 as0 as1;
```

The following table shows the update of the operand and address-stream registers when the instruction executes. The address, increment, elements, and position components of each address-stream register are shown as register fields. Fields that are updated when the instruction executes are emphasized.

	dr0	**as0**	**[.addr**	**.incr**	**.elem**	**.posn]**
add d0 as0 as1			8	4	1	3
	24		**12**	4	**2**	3
		as1	**[.addr**	**.incr**	**.elem**	**.posn]**
add d0 as0 as1			16	8	2	63
			32	8	**3**	63

Because an address-stream register specifies a limit value at which the register is updated to zero, each update actually involves an addition and comparison operation.

Updated address-stream register values are latched at the output of the update units. An updated value is not written to an address-stream register until the instruction associated with the updated value enters the write-back stage of the pipeline. The updated values are forwarded to the inputs of the update unit to allow consecutive XMU operations to receive their operands from the address-stream registers. This organization keeps all of the operand bandwidth associated with the updating of address-stream registers local to the address-stream register file and update units while ensuring that implicit writes to the address-stream registers are reflected in the architecturally exposed state of the processor when the associated instruction completes.

Register Save and Restore Operations

The memory controller provides a 64-bit interface between the processor and memory. This allows a 64-bit instruction pair to be transferred in a single cycle. It also allows save and restore instructions to load and store 64-bit index and address-stream registers as a single memory operation. If the memory interface were 32-bits wide, save and restore instructions would require multiple memory operations, during which either memory or the register would contain an inconsistent copy.

Vector Memory Operations

The vector memory operations execute in the XMU function unit. Because vector memory operations transfer multiple elements between registers and memory, vector memory operations execute across multiple cycles. The hardware that is responsible for sequencing vector memory operations is partially decoupled from the XMU pipeline, which allows the XMU to continue to execute instructions while a vector memory operation executes. However, a vector memory operation occupies the memory interface throughout the duration of its execution, which prevents other memory operations from entering the execute stage of the pipeline until the vector memory operation departs.

Interlocks and Indexed Elements

An Elm processor may continue to execute instructions while a vector memory operation transfers data between memory and the indexed registers. To simplify the implementation of the index register file, vector memory operations are given exclusive access to that half of the index register engaged in the operation: a vector load operation requires exclusive access to the write pointer in the destination index register, and a vector store operation requires exclusive access to the read pointer. An implementation could allow subsequent instructions to access an index register that is engaged in a vector memory operation by computing the effect of the vector memory operation on the index register and updating the index register when the vector memory operation begins executing. However, it is uncommon for software to need to use an index register in such a way, and when necessary, a second index register can be used to achieve the same effect. Consequently, the

Elm implementation simply stalls the pipeline when an instruction attempts to access part of an index register that is engaged in a vector memory operation constitutes until the vector memory operation completes.

To simplify software development, the execution semantics defined for vector memory operations specify that a vector memory operation appears to complete before any operations specified by subsequent instructions are performed. These execution semantics simplify the development of software and instruction scheduling because software does not need to reason about which indexed registers may be modified or read by a vector memory operation when scheduling subsequent instructions. The alternative would be to expose the decoupling of vector memory operations and to require that software explicitly synchronize the instruction stream and the completion of the vector memory operation, which would require the equivalent of a memory fence operation. The processor must ensure that an indexed register that will be written by a vector load operation is not read until after the load writes the registers, and that an indexed register that will be read by a vector memory operation is not written by a subsequent instruction until the vector memory operation has read the indexed register.

Elm uses a straightforward scoreboard to track which indexed registers may be accessed when a vector memory operation is being performed. The vector memory operation scoreboard associates a flag with each indexed register to track whether the indexed register may be accessed. When a vector load executes, the flag associated with an indexed register is set to indicate that the indexed register may be written by the operation. Similarly, when a vector store executes, the flag associated with an indexed register is set to indicate that the indexed register will be read. The flag associated with an indexed register is cleared after the vector memory operation accesses the indexed register. To reduce the energy expended updating and accessing the scoreboard, the pipeline control logic only accesses the vector memory operation scoreboard when the processor is performing a vector memory operation. We used this scoreboard architecture in Elm because most of the hardware is required to track decoupled remote loads, which we describe in Chapter 8, and therefor can be shared between the two functions. However, the hardware required by scoreboards that track individual indexed registers becomes expensive when the number of indexed registers increases.

The cost and complexity of the scoreboard can be reduced by tracking sets of indexed registers rather than individual indexed registers. The scoreboard must always ensure that the tracking of vector memory operations is conservative. For example, the size of the scoreboard can be reduced significantly if we consider the indexed registers to be partitioned into four contiguous sets and we allocate a single scoreboard entry for each quadrant of the indexed register file. The scoreboard controller would set a quadrant flag when a vector memory operation accesses an indexed register in the quadrant; similarly, pipeline control logic would delay subsequent instructions that attempt to access any of the indexed registers in a quadrant that the scoreboard indicates has a pending vector memory operation, even though the vector memory operation may not access the same indexed register. In the extreme, a single scoreboard entry can be used to indicate that a vector memory operation is in progress and that the indexed registers are not available to other instructions.

Tracking dependencies with reduced precision reduces the complexity and cost of the scoreboard, but introduces false dependencies and unnecessarily disrupts the execution of instructions when vector load operations and compute operations access nearby indexed registers. We expect that systems that provide more

indexed registers in response to greater memory access latencies will not need to track vector memory operations with a resolution of a single indexed register. As the typical memory access latency increases, software will tend to increase the number of indexed registers that are used to stage data transfers and initiate memory operations further in advance. This tends to increase the distance between where compute operations access the indexed register files and where vector memory operations access the indexed register files, allowing the scoreboard to track vector memory operations at a coarser granularity without adversely impacting processor performance.

Bandwidth Considerations

The memory controller provides a 64-bit interface between the processor and memory to allow a 64-bit instruction pair to be transferred from memory to an instruction register in one cycle. The bandwidth available between the processor and memory is sufficient to allow a vector memory operation to transfer multiple operands between memory and the indexed registers in one cycle. To enable this, the indexed register file must allow multiple 32-bit words to be read and written in one single cycle.

The write ports can be increased from 32-bits to 64-bits without significant cost, as the modifications required in the column circuits at the write interface constitute a small fraction of the register file area. Most indexed register files will have at least 64 columns, as some degree of column multiplexing is necessary to align the columns of the cell areas to the sense amplifiers at the read port. Typically, each column requires its own independent write circuits, regardless of whether the array is accessed as though it contains 32-bit words or 64-bit words, and the indexed register file already requires write-masks to support both 16-bit sub-word and 32-bit full-word writes, and supporting both 16-bit sub-word and 32-bit sub-word write masks requires the provisioning of at most a few additional wire tracks in the column circuits to route additional write enable signals that distinguish between 16-bit and 32-bit sub-word writes.

Extending the read ports from 32-bits to 64-bits is considerably more expensive when additional sense amplifiers and output latches are required. Sense amplifiers can consume a considerable fraction of the area and energy of small register file in contemporary CMOS technologies. Device mismatch and variation often require that designers use devices with longer channels to improve matching between the devices in the differential pairs of the sense amplifier. With longer channels, the device widths must be increased to achieve similar performance characteristics. Register files that use sense amplifiers implement differential read paths in which active bit-lines are partially discharged. To eliminate the need for complex clocked amplifiers, some register file architectures use single-ended read paths in which active bit-lines are completely discharged during read operations. Without clocked sense amplifiers, register files often implement clocked latch elements to hold read values stable when the bit-lines are charged in preparation for the next read operation. Without these latch elements, the output of the register file would become invalid before the next read operation begins. In either case, the increase in the area of the register file is more significant than when the width of the write ports is increased.

When considering how to provision read and write port bandwidth at the indexed register files, we should keep in mind that the vector memory operations are used to transfer operands and results that are staged

in indexed registers before being transferred to a function unit or memory. Computations that benefit from increasing bandwidth between the indexed registers and memory typically derive most of the benefit from increasing the bandwidth from memory to the indexed registers, which is used to bring operands closer to the function units. Most operations that use the indexed registers consume one or two operands and produce one result, so the operand bandwidth is greater than the result bandwidth. An additional consideration is that the vector load operations may compete with instruction loads for memory bandwidth, whereas the vector store operations have exclusive access to the memory store interface. Consequently, a reasonably balanced, though somewhat asymmetric, design option is to double the bandwidth from memory to the indexed registers to 64-bits per cycle while leaving the bandwidth from the indexed registers to memory at 32-bits per cycle.

6.3 Examples

This section presents two examples that illustrate the use of index registers and address-stream registers. In both examples, the index registers are used to access frequently referenced data stored in the indexed registers rather than memory, illustrating how index registers allow important data working sets to be captured in registers rather than memory.

Finite Impulse Response Filter

The first example we consider is a finite impulse response (FIR) filter. Finite impulse response filters are common in embedded digital signal processing applications. Finite impulse response filters are often used instead of infinite impulse response filters because they are inherently stable and, unlike infinite impulse response filters, do not require feedback and therefore do not accumulate rounding error. The following difference equation describes the output y of an $(N-1)$th order filter in terms of its input x and filter coefficients h.

$$y[n] = \sum_{i=0}^{N-1} h_i x[n-i] \tag{6.1}$$

Though a finite impulse response filter requires more computation and memory than an infinite impulse response filter with similar filter characteristics, a finite impulse response filter is readily partitioned and parallelized across multiple processor elements.

Figure 6.2 shows a code fragment that contains the relevant parts of an $(N-1)$th-order filter kernel. The input samples arrive periodically and are stored in an array that operates as a circular buffer. The filter coefficients are duplicated as illustrated in Figure 6.3 to simplify the calculation of the array indices in the inner loop of the kernel. Duplicating the coefficients provides the same effect as addressing the sample buffer as though it were a circular buffer while avoiding the additional operations required to access the array as a circular buffer, which typically involves performing a short sequence of operations to cause the index to wrap circularly. Either the samples or filter coefficients can be duplicated to simplify the addressing in the inner loop; however it is more efficient to duplicate the filter coefficients. One input sample is stored to the sample buffer during each execution of the filter, and each input sample is read N times before being discarded.

source code fragment

```
100   fixed32 sample[N];
101   fixed32 coeff[2*N];
102   forever {
103     for ( int j = 0; j < N; j++ ) {
104       sample[j] = recv();
105       fixed32 s = 0;
106       for ( int i = 0; i < N; i++ ) {
107         s += sample[i] * coeff[(j+N)-i];
108       }
109       send(s);
110     }
111   }
```

Figure 6.2 – **Finite Impulse Response Filter Kernel.** The code fragment is taken from a kernel that implements a finite impulse response filter. The input samples are stored in the `sample` array, which operates as a circular buffer. The filter coefficients are duplicated in the `coeff` array to reduce the number of operations required to compute the indices used to access the sample and coefficient arrays within the performance-critical inner loop of the kernel.

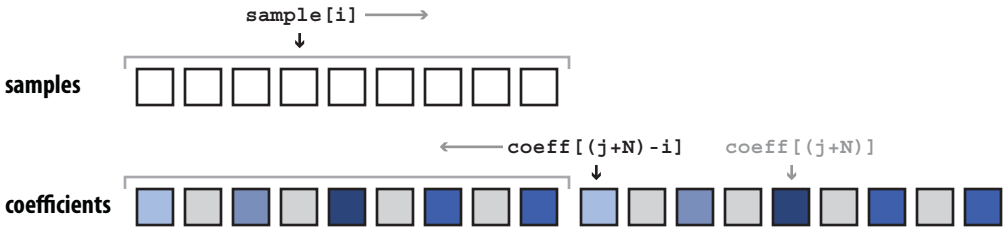


Figure 6.3 – **Finite Impulse Response Filter.** The input samples are stored in the `sample` array, which operates as a circular buffer. The coefficients are stored in the `coeff` array. Duplicating the coefficients allows the inner-loop to be implemented without circular addressing operations.

However, the filter coefficients, which control the shape of the filter response, are updated less frequently than the entries in the sample buffer. Consequently, it is more efficient to duplicate the filter coefficients rather than the samples.

We should note that the code shown in Figure 6.2 affects the order in which the additions specified by the difference equation of (7.1) are performed. This may affect the output computed by the filter when addition is not associative, for example when saturating arithmetic or floating-point arithmetic is used.

Both the filter coefficients and samples exhibit significant medium-term reuse, and together form the significant data working sets of the kernel. The **for** loop at line 106, which contains the statements that compute the output of the filter after a sample is received, dominates the execution time of the kernel and contributes most of the data references. The assembly fragment in Figure 6.4 shows the loop after scheduling and register allocation. The loop has been scheduled such that the execution time of the loop is asymptotically approaches 5 cycles per iteration.

As should be evident from the assembly fragment of Figure 6.4, the loop fetches 5 instructions pairs and loads two operands from memory during each iteration of the loop. The kernel fetches approximately

```
                alu operation                xmu operation
200    @F106:  nop                 , ld  dr0 [ar0 + ar2]     ; // load sample
201            nop                 , ld  dr1 [ar1 + ar3]     ; // load coefficient
202            nop                 , add ar2 ar2 #1          ;
203            nop                 , loop.clear pr0 [@F106]  ;
204            mac tr0 dr0 dr1 da0 , sub ar3 ar3 #1          ;
```

Figure 6.4 – Assembly Fragment Showing Scheduled Inner Loop of FIR Kernel. Sample `sample[i]` is loaded to data register `dr0`; address register `ar0` contains the address of `sample[0]`, and address register `ar2` contains the offset value `i`. Coefficient `coeff[(j+N)-i]` is loaded to data register `dr1`; address register `ar1` contains the address of `coeff[j+N]`, and address register `ar3` contains the offset value `-i`. The output value is computed using accumulator `da0`. The result of the `mac` operation, which has been scheduled in the loop delay slot, is directed to forwarding register `tr0`, where it is available when the loop completes.

$5N$ instruction pairs from instruction registers, stores one operand to memory, and loads $2N$ operands from memory during each execution.

Before explaining how indexed registers and address-stream registers allow the kernel to be executed more efficiently, we should observe that the loop can be unrolled to reduce the number of flow control and bookkeeping arithmetic instructions. When the loop is completely unrolled, the loop body can be reduced to three instructions: the **`loop`** instruction at line 203 and the **`add`** instruction can be eliminated by using the immediate offset addressing mode in the **`ld`** operation at line 200. Thus, when the loop is fully unrolled, the average execution time of a loop iteration asymptotically approaches 3 cycles. However, unrolling the loop causes the code size to expand, and eventually causes the kernel to exceed the instruction register capacity. Partially unrolling the loop reduces the execution time to somewhere between 3 and 5 cycles per iteration of the loop. Though it reduces the number of instructions executed, unrolling the inner loop does not affect the number of operands that the kernel loads from memory.

As we noted previously, the filter coefficients exhibit greater long-term reuse than the input samples, and the coefficient array elements are consequently better candidates for assignment to indexed registers. The assembly fragment in Figure 6.5 shows the inner loop of the kernel when the filter coefficients are stored in indexed registers. Promoting the filter coefficients to indexed registers reduces the length of the loop body from 5 instructions to 3 instructions. Furthermore, promoting the coefficients reduces the number of operands that are loaded from memory during each iteration of the loop from 2 operands to 1 operand. The revised kernel fetches approximately $3N$ instruction pairs from instruction registers and loads N operands from memory during each execution.

The assembly fragment in Figure 6.6 shows the inner loop when the elements of the sample array are accessed through an address-stream register. The inner loop now consists of two instructions pairs, and the revised kernel fetches approximately $2N$ instruction pairs from instruction registers and loads N operands from memory during each execution.

Image Convolution

Image convolution, or more formally two-dimension discrete convolution, is a common mathematical operation underlying many of the algorithms used in modern digital image processing applications. The discrete

```
                alu operation                xmu operation
300    @F106:  nop                 , ld  dr0 [ar0 + ar2]     ; // load sample
301            nop                 , loop.clear pr0 [@F106]  ;
302            mac tr0 dr0 xp0 da0 , add ar2 ar2 #1          ;
```

Figure 6.5 – Assembly Fragment Showing use of Index Registers in Inner Loop of fir Kernel. Sample `sample[i]` is loaded to data register `dr0`; address register `ar0` contains the address of `sample[0]`, and address register `ar2` contains the offset value `i`. The filter coefficients are stored in indexed registers, and coefficient `coeff[(j+N)-i]` is accessed through index register `xp0`. The output value is computed using accumulator `da0`. The result of the `mac` operation, which has been scheduled in the loop delay slot, is directed to forwarding register `tr0`, where it is available when the loop completes.

```
                alu operation                xmu operation
400            nop                 , ld  dr0 [as0]           ; // load sample
401    @F106:  nop                 , loop.clear pr0 [@F106]  ;
402            mac tr0 dr0 xp0 da0 , ld  dr0 [as0]           ; // load sample
403            mac tr0 dr0 xp0 da0 , nop                     ;
```

Figure 6.6 – Assembly Fragment Showing use of Address-stream Registers in Inner Loop of fir Kernel. The scheduled loop body comprises the two instruction pairs at lines 401 and 402. The instruction pair at line 400 precedes the loop body and initiates the first loop iteration, and the instruction pair at line 403 follows the loop body and completes the last loop iteration. Sample `sample[i]` is loaded to data register `dr0` using address-stream register `as0` to generate the address sequence. The filter coefficients are stored in indexed registers and accessed through index register `xp0`. The result of the `mac` operation, which has been scheduled in the loop delay slot, is directed to forwarding register `tr0`, where it is available when the loop completes.

convolution operation computes an image O from an image I and a function H according to the following expression.

$$O(y,x) = \sum_u \sum_v I(y+v, x+u) \cdot H(v,u) \tag{6.2}$$

The values of H are often referred to as the filter weights or the convolution kernel. We shall use the term filter to avoid confusion with the computation kernel that implements the convolution. The code fragment shown in Figure 6.8 illustrates the application of a convolution kernel to an input image. The code computes output image `o_img` by convolving input image `i_img` with the $M \times M$ filter `h`. The code contains two loop nests: an outer loop nest with a domain that corresponds to elements of the output image, and an inner loop nest with a domain that corresponds to the elements of the filter h. The input image is padded with an N-element wide frame so that the output of the convolution is well-defined over the entire range of the output image. Figure 6.7 illustrates the padding, and highlights the working sets associated with the computation of a single element in the output image.

To examine how address-stream registers and indexed registers allow image convolution to be performed more efficiently, we shall examine the number of arithmetic operations and memory accesses required to perform image convolution. To make the example concrete, we will assume that the convolution applies a 3×3 convolution kernel to a 256×256 image. This corresponds to setting $N = 1$, $M = 3$, and $X = Y = 256$ in the source code of Figure 6.8.

The computation required by the convolution operations is dominated by multiplication and addition

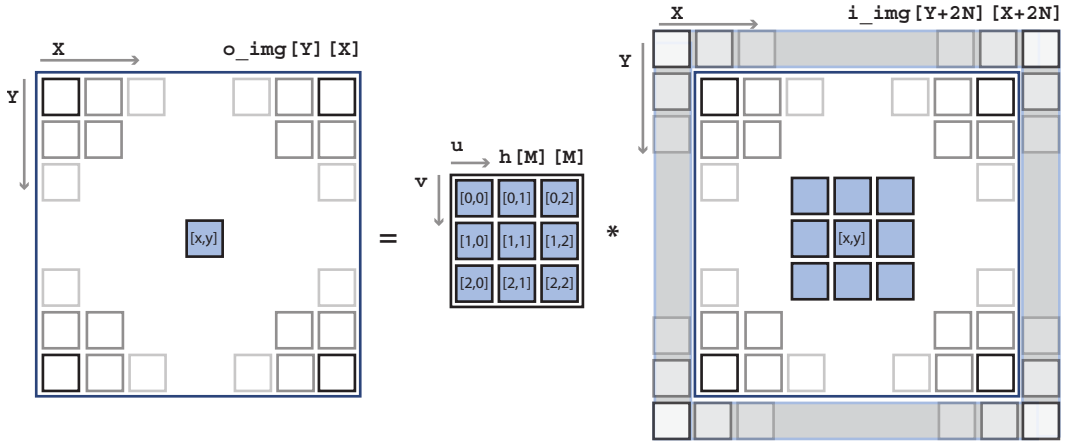


Figure 6.7 – **Illustration of Image Convolution.** Each element of the output image is computed as a weighted sum over the corresponding subdomain of the input image. The entries of h specify the weighting applied to combine elements from the input image to compute an element of the output image. The computation of an element of the output image is independent of all other output elements, and all output elements may be computed in parallel.

source code fragment

```
100   const int M = 2*N+1;
101   fixed32 i_img[Y+2*N][X+2*N];
102   fixed32 o_img[Y][X];
103   fixed32 h[M][M];

110   for ( int y = 0; y < Y; y++ ) {
111     for ( int x = 0; x < X; x++ ) {
112       fixed32 s = 0;
113       for ( int v = 0; v < M; v++ ) {
114         for ( int u = 0; u < M; u++ ) {
115           s += h[v][u] * i_img[y+v][x+u];
116         }
117       }
118       o_img[y][x]= s;
119     }
120   }
```

Figure 6.8 – **Image Convolution Kernel.** The input and output images are X elements wide and Y elements high. The input image is padded with an N-element wide frame so that the convolution is well-defined over all of the output image. Consequently the element at the upper-left corner of the input image resides at `i_img[N][N]` rather than `i_img[0][0]`. The kernel contains two loop nests: an outer loop nest defined at lines 110 – 111 with a domain that corresponds to the elements of the input image, and an inner loop nest defined at lines 113 –114 with a domain that corresponds to the elements h and the corresponding subdomain of the input image.

operations. The number of multiplications and additions required to compute the output is readily determined by inspecting the source code, and is calculated as follows.

$$(X \times Y)(M \times M) = (256 \times 256)(3 \times 3) = (65,536)(9) = 589,824 \tag{6.3}$$

Each multiplication accesses one element of the input image and one coefficient of filter h. Consequently, the aggregate number of accesses to elements of the input image and the aggregate number of accesses to elements of h are also computed by equation (6.3). Though the aggregate number of accesses are equal, there are more accesses to an individual element of h than an element of the input image, and the working sets corresponding to the element of h and the input image differ in significance. Each element of h contributes to the computation of the 65,536 elements of the output image, whereas an element of the input image only contributes the computation of 9 elements of the output image. Consequently, the working set comprising the elements of h exhibits significantly more reuse than the working sets comprising neighboring elements of the input image.

An initial implementation of the image convolution kernel is shown in Figure 6.9. As should be apparent from the assembly fragment, the implementation shown in Figure 6.9 does not use the indexed registers and address-stream registers. The inner loop is completely unrolled, and comprises lines 203 – 209. The number of instructions fetched when the code shown in Figure 6.9 executes is computed as follows.

$$256 \times (256 \times ((3 \times 8) + 4) + 2) = 1,835,520 \tag{6.4}$$

Similarly, the number of memory accesses required to fetch operands from memory and store results to memory is computed as follows.

$$256 \times (256 \times ((3 \times 6) + 1) + 0) = 1,245,184 \tag{6.5}$$

Let us first consider using address-stream registers to eliminate some of the explicit address computations. Figure 6.10 shows an implementation of the image convolution kernel that uses the address-stream registers. The use of the address-stream registers eliminates one instruction from the `for` loop at line 111, which advances the output position across a row of the input image. The number of instructions fetched when the code shown in Figure 6.10 executes is computed as follows.

$$256 \times (256 \times ((3 \times 8) + 3) + 2) = 1,769,984 \tag{6.6}$$

This corresponds to a modest 3.6% reduction in the number of instructions that are fetched. The number of operands fetched from memory remains unchanged.

Let us next consider using indexed registers to exploit reuse within the various data working sets exhibited by the image convolution kernel. Promoting the elements of the function h to indexed registers is relatively straightforward because the access pattern is affine. Figure 6.11 shows an implementation of the image convolution kernel that assigns the elements of the function h to indexed registers and uses address-stream registers to load elements of the input image and store elements of the output image. The number of instructions fetched when the code shown in Figure 6.11 executes is calculated as follows.

$$256 \times (256 \times (3 \times 5 + 3) + 2) = 1,180,160 \tag{6.7}$$

```
            alu operation                xmu operation
200         mov gr1 @i_img      , mov dr2 @h
201  @F110: mov ar0 dr2         , mov ar1 tr0
202  @F113: nop                 , ld  dr0 [ar0 + 0]      ; // dr0 = h[v][0]
203         nop                 , ld  dr1 [ar1 + 0]      ; // dr1 = i_img[y+v][x+0]
204         mac tr0 dr0 dr1 da0 , ld  dr0 [ar0 + 1]      ; // dr0 = h[u][1]
205         nop                 , ld  dr1 [ar1 + 1]      ; // dr1 = i_img[y+v][x+1]
206         mac tr0 dr0 dr1 da0 , ld  dr0 [ar0 + 2]      ; // dr0 = h[u][2]
207         add dr2 dr2 #3      , ld  dr1 [ar1 + 2]      ; // dr2 = &h[v+1][0]
208         mov ar0 tr2         , loop.clear pr0 [@F113] ; // v++
209         mac tr0 dr0 dr1 da0 , add ar1 ar1 #258       ; // ar1 = &i_img[y+v][x]
210         mov da0 #0          , st  tr0 [ar3]          ; // out[y][x] = s;
211         mov dr2 @h          , loop.clear pr1 [@F110] ; // x++
212         add gr1 gr1 #1      , add ar3 ar3 #1         ; // ar3 = &o_img[y][x]
213         nop                 , loop.clear pr2 [@F110] ; // y++
214         add gr1 gr1 #2      , nop                    ; // gr1 = &i_img[y][x]
```

Figure 6.9 – **Assembly Fragment for Convolution Kernel.** The inner-most loop is completely unrolled. The instruction pairs at lines 202 – 208 comprise the body of the unrolled loop; operand registers `dr0` and `dr0` stage the image elements and filter coefficients as they are loaded from memory; address register `ar0` supplies addresses for loading image elements, and address register `ar1` supplies addresses for loading filter coefficients. Predicate register `pr0` stores the loop with induction variable `v` and controls the execution of the loop at line 113 in the source code; predicate register `pr1` stores the loop induction variable `x` and controls the execution of the loop at line 111 predicate register `pr2` stores the loop induction variable `y` and controls the execution of the loop defined at line 110. The `add` operation at line 209 advances the pointer stored in `ar1` to the next row of the input image; the offset of 258 combines the image width (X=256) and the frame ($2N$=2). The `st` instruction at line 210 stores the value computed for the output image element; address register `ar3` provides the address of the next element in the output image. The `add` operation at line 214 advances the pointer stored in `gr1` to the next row of the input image, correcting for the padding columns introduced along the left and right edges of the input image.

```
            alu operation                xmu operation
300         nop                 , mov ar2 @i_img
301  @F110: nop                 , add ar1 ar2 as2        ; // ar1 = &i_img[y][x]
302  @F111: nop                 , add ar0 ar1 as1        ; // ar0 = &i_img[y][x]
303  @F113: nop                 , ld  dr0 [as0]          ; // dr0 = h[v][0]
304         nop                 , ld  dr1 [ar0 + 0]      ; // dr1 = i_img[y+v][x+0]
305         mac tr0 dr0 dr1 da0 , ld  dr0 [as0]          ; // dr0 = h[u][1]
306         nop                 , ld  dr1 [ar0 + 1]      ; // dr1 = i_img[y+v][x+1]
307         mac tr0 dr0 dr1 da0 , ld  dr0 [as0]          ; // dr0 = h[u][2]
308         nop                 , ld  dr1 [ar0 + 2]      ; // dr1 = i_img[y+v][x+2]
309         mac ar3 dr0 dr1 da0 , loop.clear pr0 [@F113] ; // v++
310         nop                 , add ar0 ar1 as1        ; // ar1 = &i_img[y+v][x]
311         mov da0 #0          , loop.clear pr1 [@F111] ; // x++
312         nop                 , st  ar3 [as3]          ; // out[y][x] = s;
313         nop                 , loop.clear pr2 [@F111] ; // y++
314         nop                 , add ar1 ar2 as2        ; // ar1 = &i_img[y][x]
```

Figure 6.10 – **Assembly Fragment for Convolution Kernel using Address-Stream Registers.** The assembly listing shows the code after loop unrolling.

This corresponds to a 33% reduction in the number of instructions fetched. The number of memory accesses performed when the code executes is calculated as follows.

$$256 \times (256 \times (3 \times 3 + 1) + 0) + 9 = 655,369 \tag{6.8}$$

```
                alu operation                xmu operation
400             nop                        , mov ar2 @i_img
401  @F110:     nop                        , add ar1 ar2 as2          ; // ar1 = &i_img[y][x]
402  @F111:     nop                        , add ar0 ar1 as1          ; // ar0 = &i_img[y][x]
403  @F113:     nop                        , ld  dr1 [ar0 + 0]        ; // dr1 = i_img[y+v][x+0]
404             mac tr0 xr0 dr1 da0        , ld  dr1 [ar0 + 1]        ; // dr1 = i_img[y+v][x+1]
405             mac tr0 xr0 dr1 da0        , ld  dr1 [ar0 + 2]        ; // dr1 = i_img[y+v][x+1]
406             mac ar3 xr0 dr1 da0        , loop.clear pr0 [@F113]   ; // v++
407             nop                        , add ar0 ar1 as1          ; // ar1 = &i_img[y+v][x]
408             mov da0 #0                 , loop.clear pr1 [@F111]   ; // x++
409             nop                        , st  ar3 [as3]            ; // out[y][x] = s;
410             nop                        , loop.clear pr2 [@F111]   ; // y++
411             nop                        , add ar1 ar2 as2          ; // ar1 = &i_img[y][x]
```

Figure 6.11 – Convolution Kernel using Indexed Registers. The assembly fragment shows the kernel after loop unrolling.

Promoting the elements of h to indexed registers reduces the number of memory accesses performed to load operands and store data by 47%.

To capture reuse of the image samples, we interchange the loops in the inner loop nest so that M samples from a single column are transferred to the indexed registers in each iteration of the outer loop nest. Figure 6.13 illustrates the pattern of reuse in the indexed registers. Interchanging the order of the inner loop nest complicates the computation of addresses, though the address-stream registers absorb most of the complexity. We could instead interchange the order of the loops in the outer loop nest, though doing so would affect the order in which elements of the input image would need to be streamed to the convolution kernel to avoid buffering most of the image before initiating the kernel. For example, an implementation of the image convolution kernel that proceeds down a column before advancing to the next row would need to buffer most of the input image when the input image is transferred by streaming elements along rows corresponding to horizontal scan lines.

Figure 6.12 shows the resulting assembly code fragment. The number of instructions fetched when the code executes is calculated as follows.

$$256 \times (256 \times (3 \times 3 + 3) + 4) = 787,456 \tag{6.9}$$

This corresponds to a further 33% reduction in the number of instructions fetched, and a cumulative reduction of 57% from the initial implementation. The number of memory accesses performed is calculated as follows.

$$256 \times (256 \times (3 \times 0 + 5) + 0) + 15 = 327,695 \tag{6.10}$$

Using the indexed registers to capture reuse within the elements of the input image further reduces the number of memory accesses by 50%, and achieves a cumulative reduction of 74%.

```
                alu operation                xmu operation
500             mov dr0 @i_img      , mov ar0 @i_img         ;
501             add ar0 dr0 #1      , ld.v (3) xp1 [ar0+as1] ; // load i_img[0:2][0]
502             add dr0 dr0 #2      , ld.v (3) xp1 [ar0+as1] ; // load i_img[0:2][1]
503  @F110:     mov ar0 dr0         , nop                    ; // ar0 = &i_img[y][x+2]
504  @F111:     nop                 , ld.v (3) xp1 [ar0+as1] ; // load i_img[y+0:2][x+2]
505  @F113:     mac tr0 xp0 xp1 da0 , nop                    ;
506             mac tr0 xp0 xp1 da0 , loop.clear pr0 [@F113] ; // u++
507             mac tr0 xp0 xp1 da0 , nop                    ;
508             mov da0 #0          , st  tr0 [as3]          ; // out[y][x] = s;
509             add dr0 dr0 #1      , loop.clear pr1 [@F111] ; // x++
510             mov ar0 dr0         , ld.rest xp0 [as0]      ; // ar0 = &i_img[y][x+2]
511             nop                 , loop.clear pr2 [@F110] ; // y++
512             nop                 , add dr0 ar0 as2        ; // dr0 = &i_img[y][x+2]
```

Figure 6.12 – **Convolution Kernel using Vector Loads.** The assembly fragment shows the kernel after loop unrolling. Address-stream register **as1** is configured to provide the offsets of elements in adjacent rows. The address increment field corresponds to the distance between adjacent rows in memory, and the elements field corresponds to the vector transfer size. The `ld.rest` operation at line 510 uses address-stream register `as0` to iterate through the fixed sequences of offsets that are needed to rotate the alignment the coefficients with the input samples stored in the indexed registers.

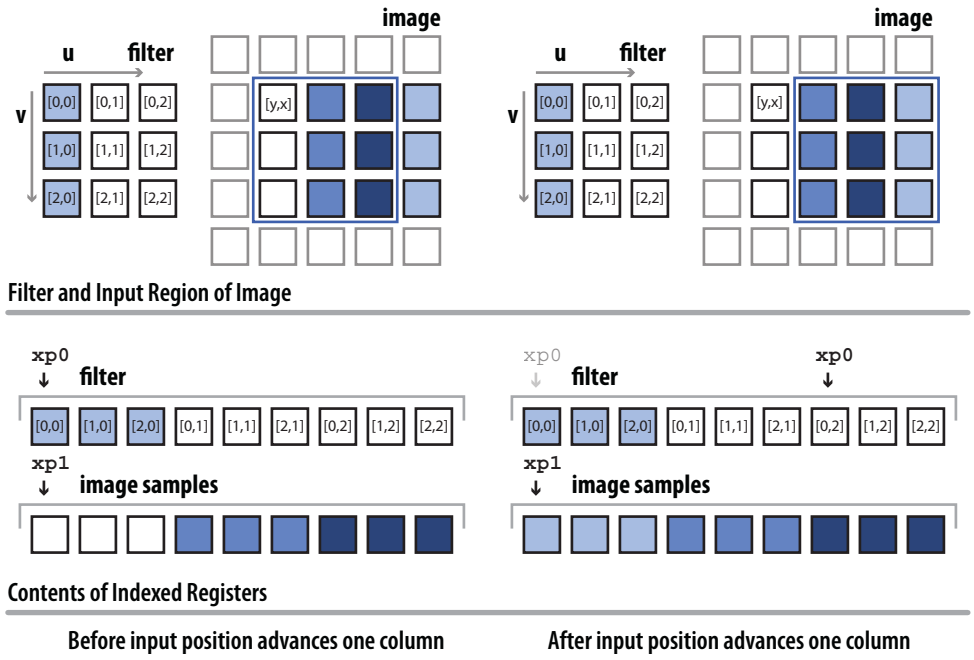


Figure 6.13 – **Data Reuse in Indexed Registers.** The contents of the indexed registers before the kernel advances the input position by one column is illustrated in the left half of the figure, and the contents after the kernel advances is illustrate in the right half. The image samples from the new column replace the samples from the column to the left of the current input range. Index register xp0 is updated to align the filter coefficients stored in the indexed registers and the image samples.

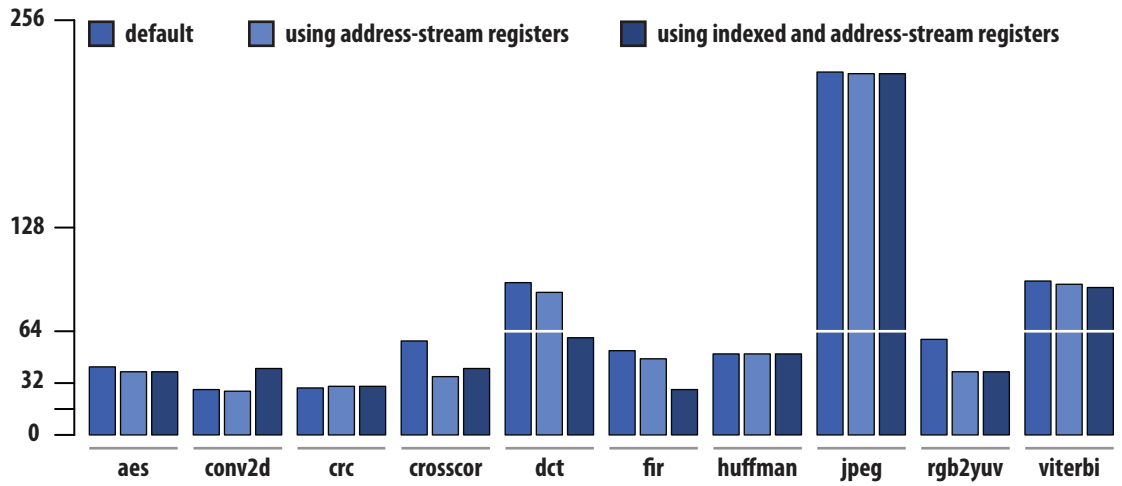


Figure 6.14 – **Static Kernel Code Size.** The static kernel code size includes load instructions that are required to initialize the index and address-stream registers when a kernel begins executing.

6.4 Evaluation and Analysis

This section provides an evaluation and analysis of the Elm indexed registers and address-stream registers. The kernels used in the evaluation were compiled using `elmcc`. We present various performance and efficiency results for three processor configurations. The default configuration uses neither address-stream registers nor index registers, and provides a reference for comparison. The configuration labeled as using address-stream registers has 4 address-stream registers in addition to the general-purpose and operand registers, but does not use the index registers. The configuration labeled as using indexed and address-stream registers has 4 index registers in addition to the 4 address-stream registers of the previous configuration.

Static Code Size

Address-stream registers and indexed registers allow software to eliminate bookkeeping and memory operations. Figure 6.14 shows the static instruction counts for each of the kernels, and illustrates the impact of compiling the kernels to use address-stream registers and indexed registers on static code. The size reported for each kernel corresponds to the number of instruction pairs in the assembly code produced by the compiler, and includes instruction pairs that are inserted into the prologue and epilogue to configure any of the address-stream and index registers used by the kernel.

Using address-stream registers consistently reduces the static kernel code size, though the significance of the reduction depends on how many bookkeeping instructions are eliminated. All of the kernels listed are able to use address-stream registers to eliminate some explicit bookkeeping instructions. For kernels in which the use of address-stream registers is limited to iterating sequentially through the input and output data buffers, such as **huffman** and **jpeg**, the elimination of a modest number of bookkeeping instructions is offset by the introduction of instructions to initialize the address-stream registers; consequently, compiling

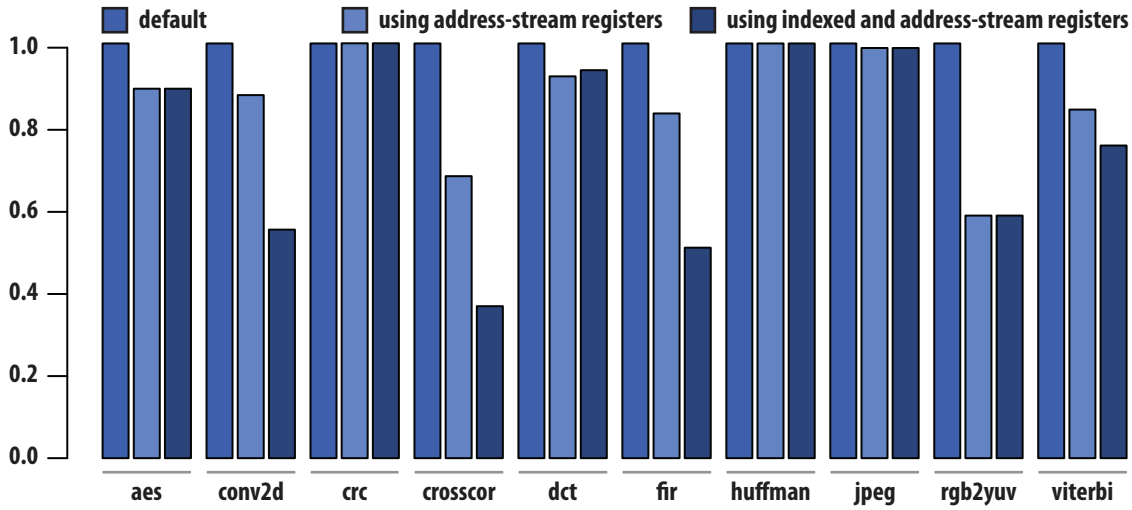


Figure 6.15 – **Dynamic Kernel Instruction Count.** The dynamic kernel instruction includes instructions that are executed to initialize the index and address-stream registers.

the kernels to use address-stream registers provides a modest reduction in the static code size. For kernels in which address-stream registers are used to eliminate bookkeeping instructions throughout the kernel, such as **crosscor** and **rgb2yuv**, eliminating bookkeeping instructions reduces the static kernel code size by 33% to 40%.

Using index and indexed registers consistently eliminates load and store instructions in those kernels with working sets that can be captured in the indexed registers; however, the additional instructions required to initialize and manage the index registers partially offsets the reduction in the code size. The increase in the static code size of the **conv2d** and **crosscor** kernels results from restructuring the kernels to enhance the use of the indexed registers. To allow samples from the input image to be stored in indexed registers, the order of the loops that implement the filter computation in the **conv2d** kernel is interchanged so that samples in a column are loaded to adjacent indexed registers. This reduces the number of times each sample is loaded, but requires additional address-stream registers, which must be initialized when in the kernel prologue. The increase in the static code size of the **crosscor** kernel results from restructuring the computation to compute multiple cross-correlation coefficients in parallel. The initial version computes a single cross-correlation coefficient, storing the coefficient being calculated in a dedicated accumulator and using multiply-accumulate operations to update the coefficient. Using the indexed registers to compute multiple coefficients in parallel allows an input sample to be incorporated into multiple coefficients after being loaded, reducing the number of times each input sample is loaded. For both kernels, restructuring the computation introduces additional instructions outside the important loops, which accounts for the increase in the static code size.

Dynamic Instruction Count

Figure 6.15 shows the normalized dynamic instruction counts, and illustrates the ability of address-stream registers and indexed registers to eliminate bookkeeping operations from a computation. The dynamic instruction count includes those instruction pairs that are executed to initialize and configure the address-stream and index registers.

As should be apparent from comparing Figure 6.14 and Figure 6.15, the reduction in the dynamic instruction count is typically greater than the reduction in the static instruction. This should be expected. Because the static code sizes of the kernels are relatively small, introducing a few additional instructions to initialize and configure the address-stream and index registers produces a noticeable increase in the sizes of the kernels. However, the additional instructions are executed infrequently and consequently comprise a small fraction of the dynamic instruction count.

Throughput and Latency

Figure 6.16 shows the normalized kernel execution times. As Figure 6.16 illustrates, using address-stream and indexed registers typically reduces kernel execution time and improves throughput. Computations with relatively few bookkeeping instructions and working sets that cannot benefit from indexed registers often exhibit small, almost negligible, improvements. The **crc** and **huffman** kernels are representative of such computations. Computations with working sets that can be captured in indexed registers often exhibit significant performance improvements when compiled to use address-stream and indexed registers. For example, the execution times of the **conv2d**, **crosscor**, and **fir** kernels are dominated by such computations, and the throughput of these kernels increases by $1.7\times$ to $2.7\times$.

Though the kernel execution times are qualitatively similar to the dynamic instruction counts shown in Figure 6.15, the execution times also include any time the processor is stalled, during which instructions are not issued from the instruction registers. Vector memory operations may introduce additional stall cycles without affecting the execution time of a kernel: the additional stall cycles correspond to time that would be spent executing an equivalent sequence of memory operations were a vector memory operation not used. These additional stall cycles appear because a processor may only have one outstanding vector memory operation and cannot execute other memory operations when a vector memory operation is in progress. Consequently, attempting to execute a memory operation when a vector memory operation is in progress causes the processor to stall. If instead of scheduling a vector memory operation, software scheduled an equivalent sequence of memory operations, these stall cycles would be occupied by memory operations. Ignoring interference from other processors competing for memory bandwidth, the number of cycles required to perform the vector memory operations will not exceed the time spent executing memory operations, as the execution time depends on the number of memory operations that are performed. However, a vector memory operation reduces the number of instructions that are issued.

Though not present in the execution times reported in Figure 6.16, using indexed registers instead of memory to store operands may provide additional performance improvements by reducing contention in the memory system. Typically, multiple processors compete for access to the memory system and memory

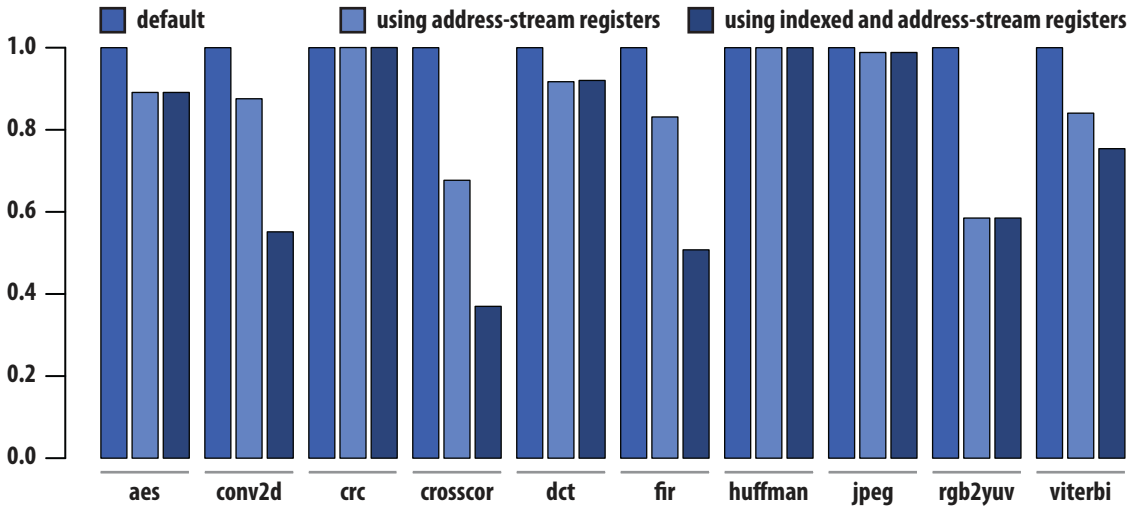


Figure 6.16 – **Normalized Kernel Execution Times.** The execution times of each kernel is normalized to the baseline register organization.

bandwidth. When the memory system is unable to complete operations attempted by different processors in parallel, the memory system serializes incompatible operations. The serialization increases the effective latency of those memory operations that are deferred. In general, it is difficult for software to anticipate when such conflicts will occur and to schedule memory operations accordingly because the conflicts result from operations performed by software executing in parallel on different processors. Using the indexed registers instead of memory to store frequently accessed data reduces demand for memory bandwidth, which reduces the likelihood of collisions in the memory system. Indexed registers also provide predictable access latencies, which simplifies the process of compiling and scheduling software to satisfy real-time performance constraints.

Instruction Delivery Energy

Indexed and address-stream registers improve energy efficiency by eliminating bookkeeping instructions from the instruction stream, which accounts for the reduction in the dynamic instruction counts of Figure 6.15. This allows machines such as Elm to deliver a greater fraction of instructions from instruction registers because it reduces instruction register pressure. Instructions in the prologue of a kernel that configure the index and address-stream registers used by the kernel are typically executed once, when the kernel loads. Consequently, these instructions do not contribute significantly to instruction register pressure, though they must be loaded from the backing memory when execution transfers to a kernel.

Figure 6.17 shows the energy expended delivering instructions, along with the fraction of the energy consumed loading instructions from the local memory. The reported energy is normalized within each kernel.

For most kernels of the kernels shown in Figure 6.17, the reduction in the instruction delivery energy simply reflects the reduction in the dynamic instruction count. The **dct** kernel is a notable exception, as

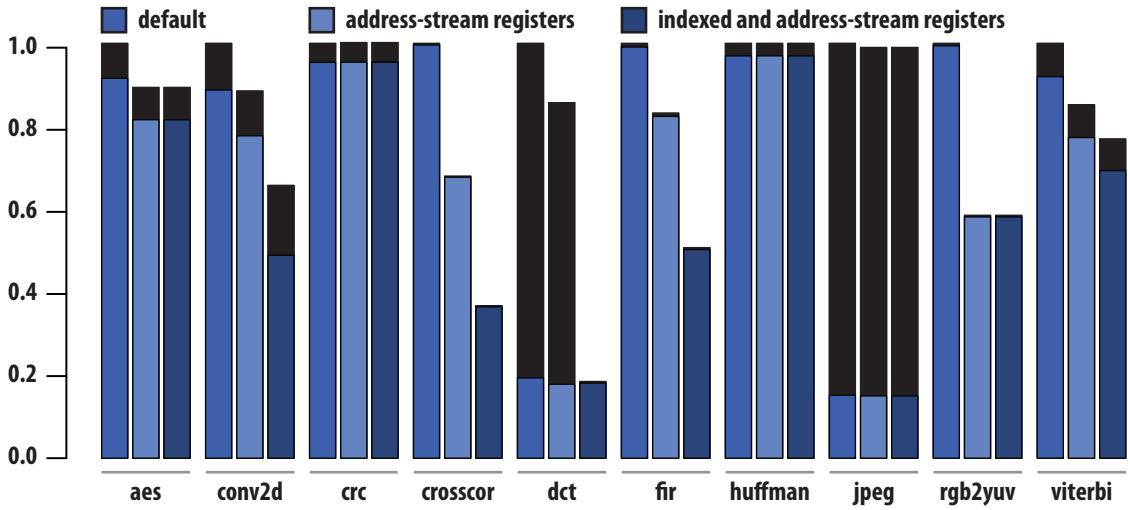


Figure 6.17 – Normalized Kernel Instruction Energy. The energy consumed staging instructions in instruction registers and memory is normalized within each kernel. The black component of each bar corresponds to the energy consumed transferring instructions to instruction registers, and includes both the energy consumed accessing the backing memory and storing the instruction to the instruction registers.

it exhibits the most significant reduction. The address-stream and index registers allow a single loop to implement both the horizontal and vertical phases of the **dct** kernel. An address-stream register is used to load the samples from memory. The address-stream register is updated at the start of the horizontal phase to generate addresses that traverse rows of the input data, and is updated again at the start of the vertical phase to generate addresses that traverse columns of the input data. The constants used to compute the transform values are stored in indexed registers and accessed through index registers. The resulting reduction in the instruction count is sufficient to allow the entire kernel to fit in the instruction registers (Figure 6.14).

Operand Delivery Energy

Indexed registers reduce operand delivery energy by replacing expensive memory accesses with less expensive register file accesses. Though less significant, they also reduce the number of address operands read from register files by memory operations to compute the effective address of load and store operations. Address-stream registers reduce operand delivery energy by replacing general-purpose register accesses with less expensive operand register accesses.

Figure 6.18 shows the energy expended delivering operands and staging results for each of the kernels. The address-stream registers are included in the operand register energy. The address-stream register energy includes the energy expended reading a register, computing the next address in the sequence, and writing the updated value to the register. Integrating the address-stream registers into the datapath of the XMU requires additional multiplexer ports that increase the address registers access energy. The index register access energy appears in the general-purpose register file component, as the indexed registers are implemented in

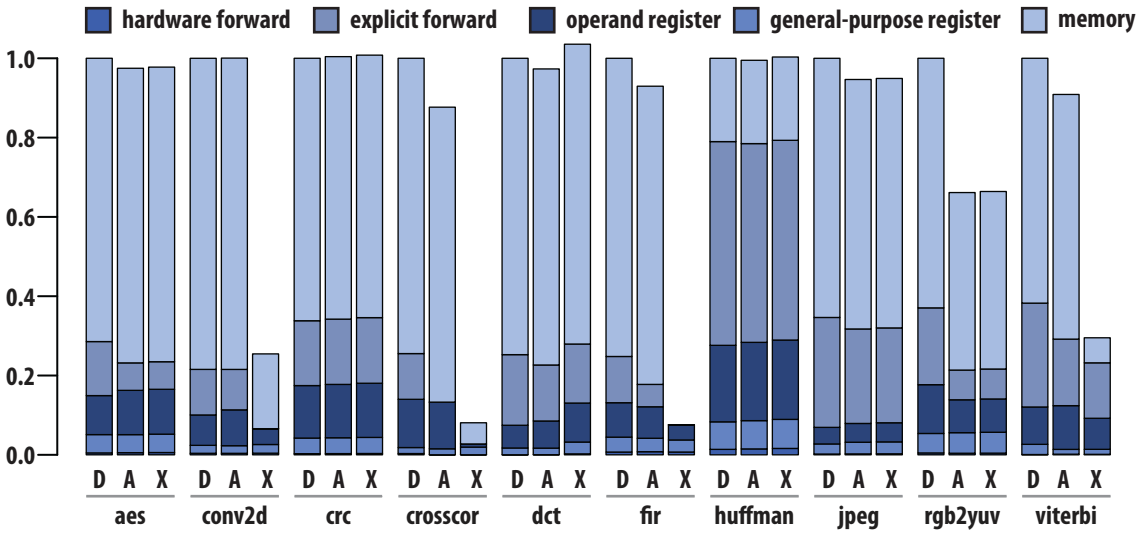


Figure 6.18 – **Normalized Kernel Data Energy.** The energy consumed staging data in registers and memory is normalized within each kernel. The three bars within each kernel, from left to right, correspond to the baseline register organization; the baseline organization extended with address-stream registers; and the baseline organization extended with both address-stream registers and index registers.

the general-purpose register file. The index register access energy includes the energy expended reading the index registers, updating the index, and writing the updated index back to the index register. Integrating the index registers and the general-purpose register file requires additional multiplexers and logic to control the selection of the addresses sent to the general-purpose register file. The additional logic increases the cost of accessing operands stored in general-purpose registers.

The most significant reductions in operand energy arise when operands are assigned to indexed registers instead of memory. For example, the index registers provide a 91% reduction in operand energy for the **crosscor** kernel beyond the 12% reduction provided by the address-stream registers. Similarly, the indexed registers provide a 92% reduction for the **fir** kernel beyond the 7% reduction provided by the address-stream registers. As the operand energy is dominated by memory accesses without the indexed registers, indexed registers would provide greater reductions were a more expensive memory used as the local memory. For example, the reduction would be greater were a cache memory or a software-managed memory of greater capacity used as a local memory.

The address-stream registers reduce the energy expended delivering operands for the **aes** and **jpeg** kernels by allowing additional operands to be stored in operand registers rather than more expensive general-purpose registers. The increase in the operand energy incurred in the indexed register configuration results from the increased cost of accessing the general-purpose registers.

6.5 Related Work

Vector processors [123] provide large collections of physical registers that are accessed indirectly as elements of vector registers [10, 59]. Because many important kernels in embedded and multimedia applications are vectorizable, the register organizations used in vector processors are able to capture a significant fraction of intra-kernel reuse and locality in embedded applications [84]. Contemporary vector-processor architectures support both vector and multithreaded execution models [88, 120]. For example, the Scale Vector-Thread architecture [88] supports both vector and multithread compute models using a collection of processors to implement a vector of virtual processors when operating as a vector processor. Like Elm, the Scale Vector-Thread architecture provides both registers that are accessed directly (shared registers) and registers that are accessed indirectly (private registers) using virtual register specifiers, and allows software to configure the allocation of physical registers between the sets of directly and indirectly accessed registers. Contemporary graphics processing unit architectures also allow physical registers to be dynamically partitioned among thread contexts, though typically this is managed by dedicated hardware or device-level software [41].

Stream processors such as Imagine [121] provide a large software-managed stream register file that is accessed indirectly through stream buffers. The stream register file is partitioned among multiple active streams, and software controls the allocation of entries in the stream register file to streams. The stream register file is used to stage bulk data transfers between memory and local registers distributed among the function units. It is also used to temporarily store elements of streams transferred between computation kernels to avoid transferring intermediate data to memory. This allows a stream register file to be used to capture inter-kernel producer-consumer locality, and reduces off-chip memory bandwidth. Because a stream register file must be large enough to stage data between memory and the stream processor, stream register files are much larger than the indexed register files used in Elm.

The early reduced instruction set computer (RISC) architectures [114] used register windows to provide software access to more physical registers than could be encoded directly in an instruction. Register window organizations typically partition the physical registers into a set of global registers and multiple sets of overlapping windows. Register windows provide software with the abstraction of an unbounded stack of windows, though only a subset near the top of the stack are maintained in physical registers; the remaining windows must be stored in memory. A distinguished control register determines which of the physical register window is active. Procedure call and return instructions implicitly update the active window, with call instructions allocating a new set of registers for use by the called procedure. Should the physical registers assigned to the new window be in use when a call instruction executes, the contents of the registers are transferred to memory and must later be restored. Privileged software may explicitly modify the active register window, typically to end a spill or fill trap handler. Register windows continue to be used in contemporary UltraSPARC architectures [104].

Itanium implements hardware-managed register stacks [28]. The register stack mechanism is similar to the register window scheme described previously. The architecture partitions the general register file into a static and stacked subset. The static subset is visible to all procedures; the stacked subset is local to each procedure and may vary in size from 0 to 96 registers. The register stack mechanism is implemented by

renaming register addresses during procedure call and return operations. Registers in the register stack are automatically saved and restored by a hardware register stack engine.

In addition to register stacks, Itanium implements a rotating register file mechanism for use in software-pipelined loops [118, 119]. A single architectural register in the rotating portion of the register file may be used in place of multiple architectural registers to specify instances of a variable defined by different iterations of a loop. Equivalent software techniques such as modulo variable expansion [93] introduce a unique name for each simultaneously live instance of a variable so that values defined in different iterations of a loop can be distinguished. This increases architectural register requirements and expands code size, as a sufficient number of interleaved iterations of the loop body must be introduced to accommodate the naming constraints of the software-pipelined loop.

Itanium allows a fixed-sized region of the floating-point register files and a programmable-sized region of the general register file to rotate when a software-pipelined loop type branch executes. The rotation is implemented by renaming register addresses based on the value of rotating register base field contained in a control register. Each rotation decrements the contents of the affected rotating register base values modulo the size of their respective software-defined rotating regions. Aside from the change of the rotating register base values, the rotating register mechanism is not visible to software.

Along with other features that distinguish explicitly parallel instruction computing (EPIC) architectures such as Itanium from their VLIW ancestors, rotating registers were introduced to improve the performance of processors in which the compiler explicitly exploits instruction-level parallelism. Rotating registers were originally used in the Hewlett-Packard Laboratories PlayDoh architecture to improve the code density of software-pipelined loops [76]. These early EPIC style architectures originated as parametric processor architectures, developed to enable an embedded system design paradigm in which a compiler generates an application-specific processor for an embedded application [126]. This allows a system designer to describe an embedded system in a high-level programming language. Essentially, in addition to compiling an application, the compiler generates a set of specific bindings for processor parameters. Similar strategies have been adapted for designing customized chip-multiprocessors [136] and digital signal processors for embedded systems from descriptions of applications provided in high-level programming languages; for example, the AnySP signal processor relies on a similar strategy of customizing hardware for a specific application to improve efficiency [150].

Register queues provide an alternative mechanism to rotating registers for decoupling physical registers from architectural register names so that more physical registers can be provisioned, improving the applicability of software-pipelining [132]. Register queue architectures let software construct queues in the register file to buffer multiple live instances of a variable introduced during modulo variable expansion. Register queue architectures require hardware tables for recording the mapping of architectural register names to physical registers and register queues, and for mapping each appearance of an architectural register that has been mapped to a register queue to an appropriate offset within its associated queue. Index and indexed registers can be used to achieve a similar decoupling of physical registers and architectural register names, to partition physical registers among architectural register names, and to implement queues for decoupling memory operations.

In contemporary computer systems, software pipelining delivers greater performance improvements when used to tolerate memory access latencies rather than arithmetic unit latencies, which are short compared to typical memory access times. The number of physical registers required for a compiler to software pipeline a loop effectively increases as the product of the number of variables in the loop and the number of interleaved loop iterations. The additional physical registers provided by a rotating register file allow software to schedule more interleaved loop iterations. As the number of interleaved iterations required to tolerate the latency of operations in a loop increases with the operation latencies, additional physical registers allow software pipelining to tolerate longer function unit latencies. In addition to tolerating memory access latency, indexed registers let software effectively capture medium-term reuse and locality in local register files rather than memory, which is considerably more difficult to achieve using rotating register files and register queues.

The VICTORIA [37] architecture defines a set of extensions to VMX, the SIMD extensions to the PowerPC architecture [27], that uses indirection to increase the number of VMX registers that are available to software. The indirect-VMX (iVMX) extensions allow a processor to implement as many as 4,096 VMX registers. The extensions define several sets of map registers that store offsets into the VMX register file. The map registers are organized as tables that are indexed using the VMX register specifier encoded in an instruction. An independent map table is maintained for each of the 4 register specifier fields defined by the VMX extensions. The effective address of a VMX register is computed by adding the offset stored in the associated map register and the address encoded in the register specifier field of an instruction. Software is responsible for managing the map registers, and must explicitly modify map registers to access a different subset of the extended VMX registers, which limits the number of VMX registers that can be accessed without executing bookkeeping instructions that modify the map registers. However, the extensions preserve backwards compatibility with the original VMX extensions.

The eLite digital signal processor [106] provides a collection of vector pointer registers that are used to index a set of vector element registers. Like the index registers described in this chapter, the vector pointer registers are automatically updated when used to access the vector element registers. However, the eLite architecture supports a single-instruction multiple-data SIMD execution model in which instructions operate on disjoint data that are packed into short fixed-width vectors. The architecture uses a distinguished set of vector accumulator registers to receive the results of most vector operations, with the perhaps obvious exception of operations that transfer vector accumulator registers to vector element registers and to memory. The vector element registers are organized to allow data in vector element registers to be arbitrarily packed into operand vectors as they are read from the vector element register file. Every access to the vector element registers is performed indirectly through the vector pointer registers, and each of the 4 elements in a packed operand may use a different pointer value to access the vector element register file. This supports a vector programming model in which data in memory and in the vector accumulator registers are accessed as packed vector elements, while data stored in the vector element registers are accessed as scalar elements that are packed into vector operands on access. The creators of the eLite architecture refer to this as a register organization that supports single-instruction multiple disjoint data SIMdD operations, as opposed to the conventional single-instruction multiple packed data SIMpD multimedia instruction sets found in commodity processors. The eLite register organization allows data reuse that crosses vector lanes to be captured in the

vector element register file without performing explicit permutation operations or memory operations to align vector elements. However, it does so by introducing additional register file ports. Each vector lane has its own dedicated read and write ports to the vector element register file. Though the impact of the additional ports on the area and access energy of the vector register file organization is not quantified, the additional ports render the vector element registers less area-efficient and less energy-efficient than a conventional SIMD register organization[122]. The register organization also complicates the design and implementation of the forwarding network, which must detect when a result is forwarded across vector lanes based on the pointer values delivered from the vector pointer registers.

Some general-purpose architectures support addressing modes that update one of the registers used to compute the effective memory address as a side effect of load and store operations[60]. The PowerPC architecture is a representative example. It provides both a register index addressing mode that updates the base register used in the operation with the effective address, and an immediate addressing mode that similarly updates the base register [26]. Automatic update addressing modes lack much of the flexibility of address-stream registers. Address-stream registers allow both the sequence of addresses in the address stream and the state of the stream to be compactly and efficiently encoded. By providing explicit named state, address-stream registers allow software to save and restore the complete state of an address stream. Address-stream registers also decouple the updating of the address stream sequence from the computed effective address, and allow any instruction that can access an address-stream register to consume and update the address sequence.

The PowerPC architecture also provides load and store operations that transfer multiple words between general-purpose registers and memory. Because the range of registers that is affected by the operation is explicitly encoded in instruction, the use of these operations is typically limited to moving blocks of registers between the stack and the register file at procedure call boundaries, and to loading and storing aggregate data types.

6.6 Chapter Summary

Indexed registers expose mechanisms for accessing registers indirectly. The resulting register organization allows software to capture reuse and locality in registers when data access patterns are well structured, which improves the efficiency at which data are delivered to function units. Address-stream registers expose hardware for computing structured address streams as address stream registers. Address stream registers improve efficiency by eliminating address computations from the instruction stream, and improve performance by allowing address computations to proceed in parallel with the execution of independent operations.

Chapter 7

Ensembles of Processors

This chapter introduces the Ensembles of processors organization used in an Elm system. An Ensemble is a coupled collection of processors, memories, and communication resources. An Elm system is a collection of Ensembles. The Ensembles share a distributed memory system, and external interfaces for communicating with other chips. The Ensemble is the fundamental compute design unit in an Elm system, and is extensively replicated throughout an Elm system. This extensive replication and reuse of a small design unit allows aggressive custom circuits and design efforts to be used effectively within an Ensemble.

7.1 Concept

Figure 7.1 illustrates the organization of the processors and memories comprising an Ensemble. The processors in an Ensemble are assigned a unique processor identifier in the range [0,3]. We usually refer to the processors using the names EP0, EP1, EP2, and EP3 when we need to identify a particular processor.

Ensemble Memory

The Ensemble memory is a local software-managed instruction and data store that is shared by the processors in the Ensemble. The Ensemble memory serves several important functions. Perhaps most importantly, it provides local memory for storing important instruction and data working sets that exceed the capacity of the register files close to the processors. The Elm compiler uses the local Ensemble to store instructions that may be loaded to the instruction registers during the execution of a kernel, which reduces the performance and energy impact of loading instructions from memory during the execution of a kernel. The Elm compiler also uses the local Ensemble memory to temporarily spill data when it is unable to allocate registers to all of the live variables in a kernel.

The Ensemble memory also provides local storage for staging data and instruction transfers between the processors in the Ensemble and memory beyond the Ensemble. The storage capacity provided by the local Ensemble memory allows software to use techniques such as prefetching and double buffering that exploit

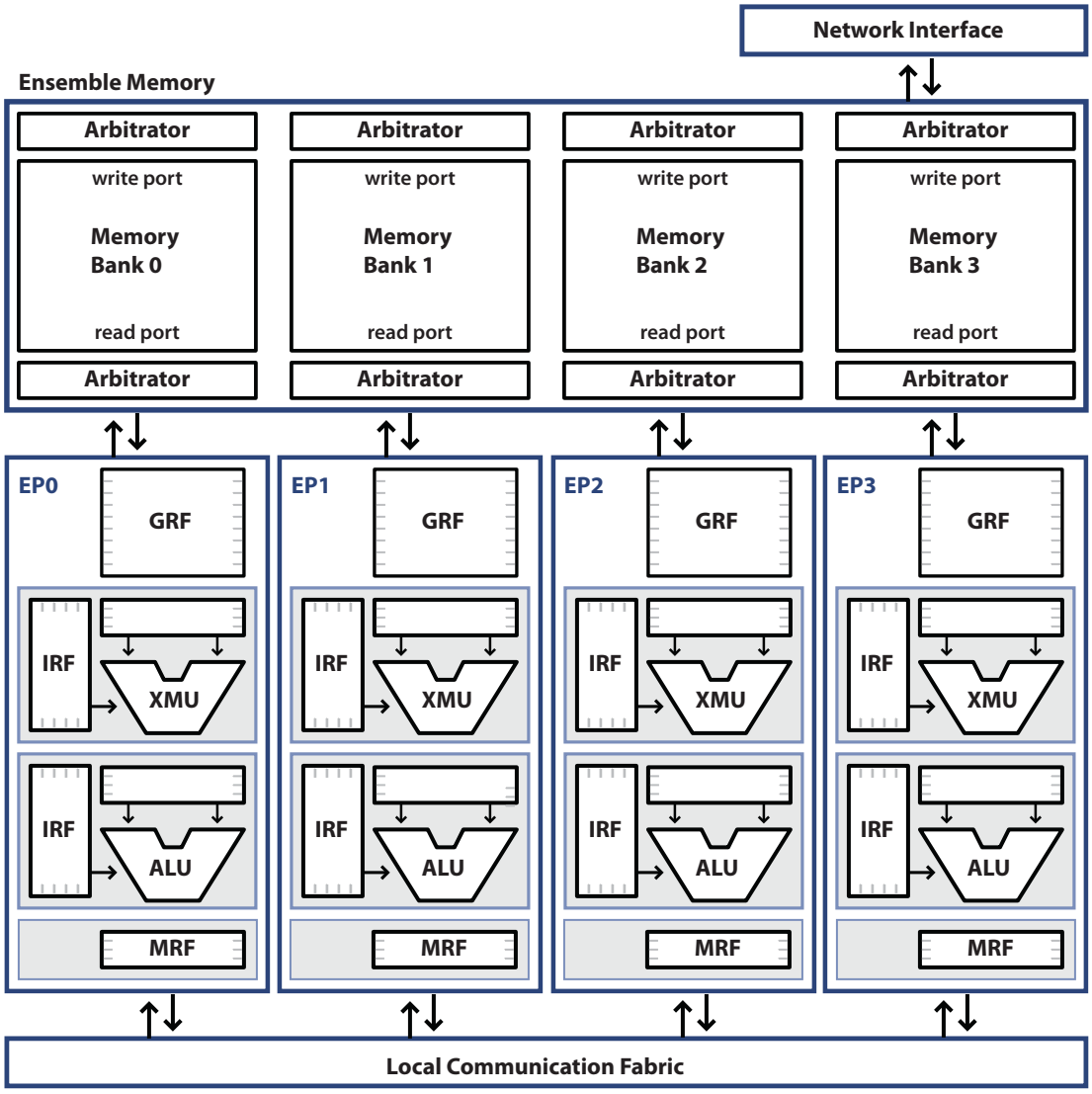


Figure 7.1 – **Ensemble of Processors.** The Ensemble organization allows the expense of transferring instructions and data to an Ensemble to be amortized when multiple processors within the Ensemble access the instructions and data. The Ensemble memory is a local software-managed instruction and data store that is shared by the processors in the Ensemble. The local communication fabric provides a low-latency, high-bandwidth interconnect for efficiently transferring data between threads executing concurrently on different processors. These links may be used to stream data between threads that are mapped to different processors within an Ensemble, avoiding the expense of streaming data through memory.

communication and computation concurrency to effectively hide remote memory access latencies.

The Ensemble organization allows the expense of transferring instructions and data to an Ensemble to be amortized when multiple processors within the Ensemble access the instructions and data. This provides an

important mechanism for significantly improving the efficiency of instruction and data delivery when data parallel codes are mapped to the processors in an Ensemble.

The Ensemble memory is banked to allow it to service multiple transactions concurrently. Each processor within an Ensemble is assigned a preferred bank. Accesses by a processor to its preferred bank are prioritized ahead of accesses by other processors and the network interface. Instructions and data that are private to a single processor may be stored in its preferred bank to provide deterministic access times. The arbiters that control access to the read and write ports are biased to establish an affinity between processors and memory banks. This allows software to make stronger assumptions about bandwidth availability and latency when accessing data that may by shared by multiple processors.

Local Communication Fabric

Threads executing in an Ensemble may communicate by reading and writing shared variables located in the local Ensemble memory. However, communicating through memory requires that threads execute expensive load and store instructions. The local communication fabric provides a low-latency, high-bandwidth interconnect for efficiently transferring data between threads executing concurrently on different processors. These links may be used to stream data between threads that are mapped to different processors within an Ensemble, avoiding the expense of streaming data through memory. The communication links also provide an implicit synchronization mechanism. This allows the local communication fabric to be used to transport operands between instructions from a single thread that are partitioned and mapped across multiple processors within an Ensemble. It also allows software executing on different processors to use the communication fabric to synchronize execution.

In addition to the dedicated point-to-point links that connect the processors in an Ensemble, the local communication fabric provides configurable links that can be used to connect processors in nearby Ensembles. This allows the communication fabric to be used to stream data between processors in different Ensembles, which simplifies the mapping of software to processors by relaxing some of the constraints on which processors may communicate through the local communication fabric. Because it avoids the expense of multiple memory operations, it is significantly less expensive to transfer data between processors in different Ensembles using the local communication fabric than transferring data through memory. These configurable links may be used to transport scalar operands between instructions executing in different Ensembles, though the transport latency increases with the distance the operands traverse.

Message Registers

Software accesses the local communication fabric through message registers, which provide an efficient software abstraction for the local injection and ejection ports comprising the interface to the local communication fabric. The message registers are conceptually organized as message register files, as shown in Figure 7.1. The software overheads of assembling and disassembling messages are intrinsic to message passing, as data associated with computational data structures must be copied between memory based data structures and the

network interface. These marshaling costs limit the extent to which an architecture allows software to exploit fine-grain parallelism effectively. Exposing the local communication interface as registers reduces the marshaling overheads associated with sending and receiving messages, as data may be transferred directly between an operation that produces or consumes an operand and the network interface [32]. This allows Elm to exploit very fine-grain parallelism efficiently.

Associated with each message register is a full-empty synchronization bit that allows processors to support synchronized communication through the message registers [134]. The full-empty bits allow synchronization and data access to be performed concurrently, and support efficient fine-grained producer-consumer synchronization and communication. The full-empty bit associated with a message register is set when the register is written, and is cleared when the message register is read. Most operations that access a message register inspect the state of its associated synchronization bit and use it to synchronize the access. Typically, an operation that reads a message register will only proceed when the synchronization bit is set to indicate the message register is full, and an operation that writes a message register will only proceed when the synchronization bit indicates the message register is empty. The instruction set defines distinguished unsynchronized move operations that allow software to access message registers without regard to the state of the associated synchronization bits.

The Elm architecture allows a subset of the message registers in each processor to be mapped to different links in the local communication fabric. This allows software to control the connection of processors, both within an Ensemble and across Ensembles. The configurable message registers may be used to establish additional links between processors within an Ensemble. These may be used to provision more bandwidth between processors. Perhaps more importantly, these additional links may also be used to provide additional independent named connections for transporting scalar operands between instructions, which can simplify the scheduling of instructions when multiple operands are passed between processors by relaxing constraints on the order in which operands are produced and consumed.

Links in the local communication fabric that connect processors in different Ensembles may be configured to operate as streaming links or scalar links. These links may require multiple cycles to traverse and may contain internal storage elements that operate as a distributed queue. This allows a link to contain multiple words. When a link is configured to operate as a streaming link, the synchronization bit at the injection interface will indicate the associated message register is full after the link has filled back to the injection port. This decouples the synchronization between the sender and receiver, and allows the sender to inject multiple words before the receiver accepts the first word. When a link is configured to operate as a scalar link, the synchronization bit associated with the message register is set to indicate the message register is full when the register is written. It is cleared when the acknowledgment signal, generated when the receiver reads the message register, arrives at the sender. This retains the synchronization semantics defined for operations that use the message registers to communicate within an Ensemble. However, the synchronization latency limits the rate at which data can be injected into the local communication network, and prevents the processors from fully utilizing the bandwidth provided by the communication link.

Processor Corps and Instruction Register Pools

The Elm architecture allows processors in an Ensemble to form processor corps that execute a common instruction stream from a shared pool of instruction registers. This allows the architecture to support a single-instruction multiple-data execution model that improves performance and efficiency when executing codes with structured data-level parallelism. The transition between independent execution and corps execution is explicitly orchestrated by the threads executing in the Ensemble. The structure imposed by the Ensemble organization allows the instructions that threads execute to switch between corps execution and independent execution to be implemented as low-latency operations, which allows an Ensemble to execute broad classes of data parallel tasks efficiency by interleaving the execution of thread corps and independent threads. Those parts of a task that exhibit single-instruction multiple-data parallelism may be mapped to thread corps, while those parts that exhibit less well structured parallelism may be mapped to independent threads. The ability to switch between thread corps and independent threads quickly allows programs to use independent threads to handle small regions of divergent control flow efficiently within data parallel tasks.

Mapping data parallel code to processor corps improves efficiency in several ways. Perhaps most importantly, the shared instruction register pool increases the instruction register capacity that is available to the processor corps. This allows the shared instruction register pool to capture larger instruction working sets, reducing instruction bandwidth demand at the local memory interface and the number of instructions that are loaded from memory. The efficiency improvements realized when larger instruction working sets are captured in the shared instruction registers are significant because loading an instruction from memory is considerably more expensive than accessing an instruction register file. Energy efficiency is further improved because the number of instructions that are fetched from the instruction register files is reduced when data parallel code is executed by processor corps. Instructions fetched from the shared instruction register pool are forwarded to all of the processors comprising the processor corps. Finally, because processors in a processor corps execute instructions in lock-step, synchronization operations within thread corps may be eliminated. Eliminating synchronization operations reduces the number of instructions that are executed, and may reduce the number of memory accesses that are performed.

Instruction register pools have some obvious limitations that should be considered when evaluating whether they are appropriate for a particular processor design. Transferring instructions from register files to remote processors consumes additional energy, which offsets some of the efficiency achieved by reducing the number of instruction register file accesses. In essence, instruction register pools trade instruction locality for additional capacity. Distributing the instructions introduces additional wire delays into the instruction fetch path, and contributes to the instruction fetch latency. The amount of time available to distribute instructions must be budgeted carefully, and not all architectures will have sufficient slack to allow instructions to be distributed without introducing an additional pipeline stage.

7.2 Microarchitecture

This section describes how the Elm prototype architecture issues instructions to processor corps and provides support for shared instruction registers pools. Figure 7.2 illustrates the general organization of the instruction fetch and issue hardware within a processor, with the hardware blocks that implement support for processor corps highlighted.

Pipeline Coupling

Control logic distributed through the processors and Ensemble memory system ensures that the coupled processors in a corps follow a single-instruction multiple-data execution discipline. As illustrated in Figure 7.2, remote stall signals are collected at each processor and combined to produce local pipeline interlock signals. The small physical size of an Ensemble allows control signals to be distributed among its processors quickly. To ensure that the collection and integration of remote stall signals does not introduce critical paths, the hardware that detects stall conditions is distributed through the Ensemble. For example, the arbiters that detect bank conflicts at the Ensemble memory broadcast the arbitration results to all of the processors, allowing each to determine when a bank conflicts causes a processor in the corps to stall. Similarly, the message register states, which determine when an access will cause a processor to stall, are distributed so that processors in an corps can detect when another member will stall. The pipeline control logic contains a corps register that indicates which processors belong to the same corps, which determines which remote stall signals are combined to produce the local pipeline control signals.

The instruction register organization affords fast instruction access times, and allows instructions to be distributed in the instruction fetch stage. Instructions are fetched and distributed to all of the processors in a single cycle, and the pipeline timing is the same when processors are coupled to form a processor corps. This simplifies the design of the pipeline control logic, and simplifies the programming model and instruction set architecture, as the latency of most instructions does not depend on whether an instruction is executed by an independent processor or a processor corps. The notable exception is the behavior of jump-and-link instructions, which produce a single cycle pipeline stall when executed by a processor corps. The stall is used to allow the processors to resolve which processor holds the destination instruction.

Distributed Control Flow

We use the concept of a conductor to distinguish the processor that fetches and distributes instructions to its processor corps. The conductor is identified as the processor whose instruction registers hold the next instruction to be executed by its corps. The conductor is responsible for fetching instructions from its instruction registers and forwarding it to the corps. When control transfers to an an instruction register that resides in a different processor, responsibility for conducting the corps transfers to the processor that holds the instruction.

The Elm prototype architecture uses an extended instruction pointer to address the shared instruction registers. The extended instruction pointer register (EIP) contains the most significant 2 bits of the extended

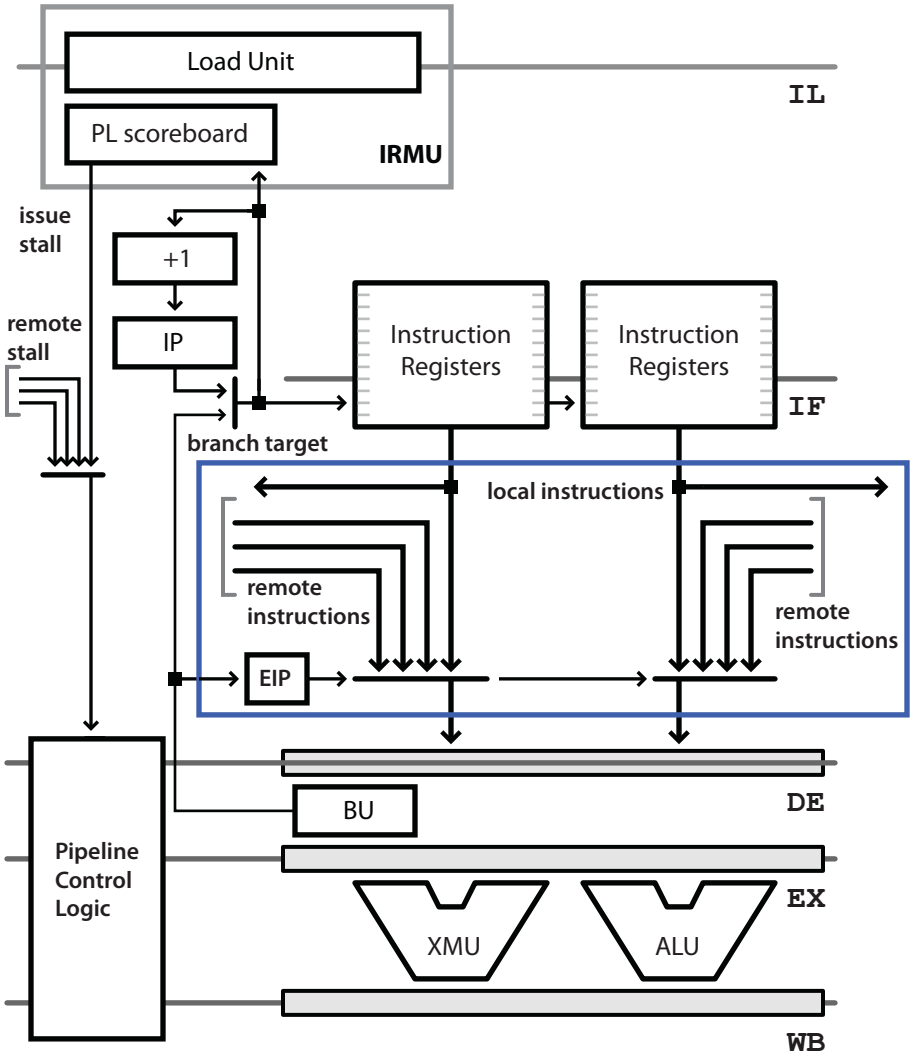


Figure 7.2 – **Group Instruction Issue Microarchitecture.** The extended instruction pointer (EIP) determines which processor holds the next instruction.

instruction pointer; the EIP register indicates which processor contains the shared instruction register that is addressed by the extended instruction pointer. The instruction pointer register (IP) contains the remainder of the extended instruction pointer. As when processors execute independently, the IP register holds the address of the next instruction to be fetched from the instruction register file.

In most situations, the conductor is the only member of the corps that needs to maintain a precise instruction pointer. The other processors need to ensure that the contents of their EIP registers are correct so that they can correctly identify the conductor. Only the conductor automatically updates its instruction pointer after each instruction is fetched. The remaining processors update their extended instruction pointers when control

flow instructions are encountered. Because control flow instructions encode their destination explicitly, an accurate extended instruction pointer value is forwarded in each control flow instruction. This ensures that all of the processors have a correct extended instruction pointer immediately following a transfer of control, and it ensures that control flow instructions always produce the required transfer of responsibility for conducting the corps.

The Elm prototype uses an inexpensive strategy for detecting when responsibility for conducting a corps should be transferred to a different processor. The conductor detects that control is about to transfer to a different processor when the hardware that updates the instruction pointer generates a carry into the extended instruction pointer bits. The carry signals that the extended instruction pointer addresses an instruction in a different processor. To transfer control, the conductor forwards its instruction pointer to the corps and signals the other members of the corps to update their extended instruction pointers. The conductor stalls the execution of the processor corps for one cycle during the transfer of control, during which the instruction forwarded by the conductor is invalid. The same strategy is used to broadcast the extended instruction pointer when the conductor identifies a jump-and-link operation.

We could avoid the one cycle stall by having the conductor detect that control is about to transfer one cycle before control actually transfers. This would require that the conductor continuously monitor the extended instruction pointer. The additional logic required to do so would toggle whenever the instruction pointer is updated, and would contribute to the overhead of delivering instructions to processors regardless of whether they operate independently or are coupled. The additional branch delay is encountered infrequently and rarely impacts application performance, as software usually inserts control flow operations to explicitly transfer control before control reaches the last instruction in the conductor's instruction register file.

Resolving Control Flow Destinations

As described in a previous chapter, Elm processors resolve the destination of control flow instructions that encode their destinations as relative displacements when instructions are transferred to instruction registers. When the instruction register management unit (IRMU) transfers a control flow instruction that encodes a relative destination to the instruction registers, it resolves the destination by adding the displacement to the shared instruction register at which the instruction is loaded. The destination is stored as a shared instruction register identifier, and the instruction is converted to an equivalent control flow instruction using a fixed destination encoding.

7.3 Examples

This section describes how the indexed registers can be used to compose messages, and presents a parallel implementation of a finite impulse response filter that uses message and processor corps.

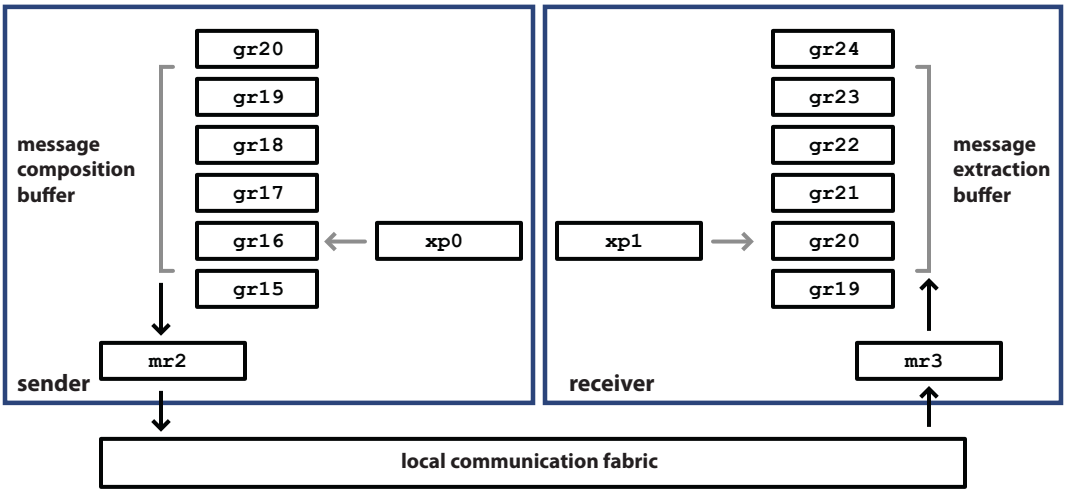


Figure 7.3 – Composing and Receiving Messages in Indexed Registers. Messages are buffered in indexed registers at the sender and receiver. The index registers are used sequence the sending and receiving of messages.

Composing and Receiving Messages in Indexed Registers

Unlike some processor architectures that provide hardware mechanisms that support efficient message passing [34, 86, 91], Elm does not provide a dedicated set of registers for composing messages. Instead, messages may be composed and received in general-purpose registers by using the index registers to sequence the transfer of messages between the general-purpose registers and message registers. This decouples the composition and extraction of a message from its transmission.

Figure 7.3 illustrates the use of general-purpose registers to compose and receive messages. Messages are composed in registers `gr16` – `gr19` and received in registers `gr20` – `gr23`. Index register `xp0` is used to send messages to message register `mr2`, and index register `xp1` is used receive messages from `mr3`.

Parallel Finite Impulse Response Filter

The following difference equation describes the output y of an order $(N-1)$ filter in terms of its input x and filter coefficients h.

$$y[n] = \sum_{i=0}^{N-1} h_i x[n-i] \tag{7.1}$$

To simplify the example, we assume that N, the number of filter coefficients, is a multiple 4, the number of processors in an Ensemble. The computation is parallelized by expanding the convolution as follows.

```
100  fixed32 sum = 0;
101  for ( int i = 0; i < N; i++) {
102    sum += h[i]*x[n+N-i];
103  }
104  y[n] = sum;
```

```
200  fixed32 sum = 0; fixed32 psum[P];
201  for ( int p = 0; p < P; p++ ) {
202    psum[p] = 0;
203    for ( int i = p*N/P; i < (p+1)*N/P; i++) {
204      psum[p] += h[i]*x[n+N-i];
205    }
206  }
207  for ( int p = 0; p < P; p++ ) {
208    sum += psum[p];
209  }
210  y[n] = sum;
```

Figure 7.4 – Source Code Fragment for Parallelized fir. The serial code fragment appears on the left. The parallel code fragment appears on the right. The data-parallel `for` loop at line 201 is mapped to `P` processors.

$$\begin{aligned}\sum_{i=0}^{N-1} h_i x[N-i] &= \sum_{i=0}^{N/4-1} h_i x[N-i] \\ &+ \sum_{i=N/4}^{2N/4-1} h_i x[N-i] \\ &+ \sum_{i=2N/4}^{3N/4-1} h_i x[N-i] \\ &+ \sum_{i=3N/4}^{4N/4-1} h_i x[N-i] \end{aligned} \tag{7.2}$$

The computation is distributed so that each thread computes one term of the expanded summation. This allows the filter coefficients and samples to be distributed across multiple processors without replicating any of the coefficients and samples.

Figure 7.5 illustrates the decomposition and mapping of the coefficients and samples to the processors in an Ensemble. Processor EP0 receives the input sample stream in message register `mr4`. Processor EP3 computes the final output of the filter and forwards it through message register `mr5`. The implementation uses indexed registers to store the filter coefficients and samples. The samples are accessed through index register `xp0` and the coefficients are accessed through index register `xp1`. The message registers are configured to allow samples to be forwarded through message register `mr5` and received in message register `mr4`.

Figure 7.6 shows the assembly code fragment that implements the kernel of the finite impulse response filter. The code that configures the processors and loads the instruction registers are not shown to simplify and focus the example. Basic block `BB1` is executed by a processor corps comprising all four of the processors in the Ensemble. The instructions in basic block `BB1` implement the multiplications and additions that form the kernel of the filter computation. The partial sums computed by the individual processors are sent to processor EP3 at line 116, where the multiply-and-accumulate operation writes message register `mr3`.

Basic block `BB2` is executed by all of the processors. The instructions in basic block `BB2` forward samples

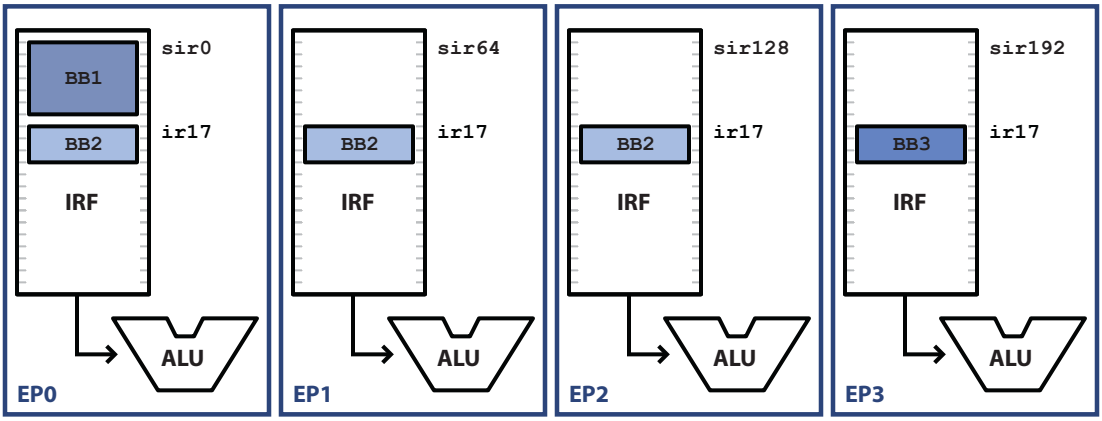


Figure 7.5 – **Mapping of fir Data, Computation, and Communication to an Ensemble.** The samples are accessed through index register `xp0`, and the filter coefficients are accessed through index register `xp1`. The message registers are configured to allow samples to be forwarded through message register `mr5` and received in message register `mr4`. Processor EP0 accepts input samples through message register `mr4`, and processor EP3 forwards the output of the filter to message register `mr5`.

```
100    .begin-iblock @IB1
101  ┌ BB1:  mac zr0 da0 xp0 xp1 , nop ;                              ┐
102  │       mac zr0 da0 xp0 xp1 , nop ;                              │
103  │       mac zr0 da0 xp0 xp1 , nop ;                              │
104  │       mac zr0 da0 xp0 xp1 , nop ;                              │
105  │       mac zr0 da0 xp0 xp1 , nop ;                              │
106  │       mac zr0 da0 xp0 xp1 , nop ;                              │
107  │       mac zr0 da0 xp0 xp1 , nop ;                              │
108  │       mac zr0 da0 xp0 xp1 , nop ;                              │
109  │       mac zr0 da0 xp0 xp1 , nop ;                              │
110  │       mac zr0 da0 xp0 xp1 , nop ;                              │
111  │       mac zr0 da0 xp0 xp1 , nop ;                              │
112  │       mac zr0 da0 xp0 xp1 , nop ;                              │
113  │       mac zr0 da0 xp0 xp1 , nop ;                              │
114  │       mac zr0 da0 xp0 xp1 , nop ;                              │
115  │       mac zr0 da0 xp0 xp1 , nop ;                              │
116  │       mac mr3 da0 xp0 xp1 , nop ;                              │
117  └       mov da0 zr0         , jmp.i [ir17] ;                   │
118    .begin-iblock @IB2                                            │
119  ┌ BB2:  mov mr5 xp0         , nop ;                            ┐ IB2
120  └       mov xp0 mr4         , jmp.c [sir0] ( ep0:ep3 ) ;       ┘ │
121   .end-iblock @IB1, IB2                                          │
122   .begin-iblock @IB3                                             │
123  ┌ BB3:  add dr0 mr0 mr1     , mov xp0 mr4 ;                    ┐ │
124  │       add dr1 mr2 mr3     , mov zr1 xp0 ;                    │ │
125  └       add mr5 dr0 dr1     , jmg.c [sir0] ( ep0:ep3 ) ;       ┘ IB3
126   .end-iblock @IB3                                               │
127                                                                  ┘
```

Figure 7.6 – **Assembly Listing of Parallel fir Kernel.**

between the processors. The oldest input sample is read through index register `xp0` and written to message register `mr5` at line 119. The transferred sample value is received in message register `mr4` at line 120 and written to an indexed register through index register `xp0`.

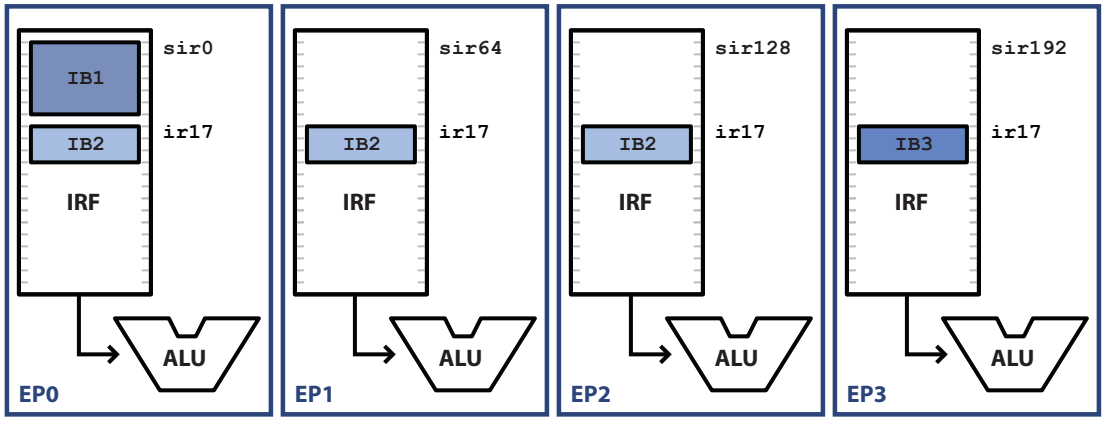


Figure 7.7 – Mapping of Instruction Blocks to Instruction Registers.

Basic block BB3 is executed by processor EP3. The instructions in basic block BB3 compute the output of the filter by combining the partial sums sent from the processors. The partial sums are received in message registers mr0 – mr3. The filter output is computed at line 124 and forwarded to message register mr5. The jump as processor corps instructions at lines 120 and 126 transfer control back to basic block BB1.

Figure 7.7 shows the mapping of instruction blocks to instruction registers. The instruction blocks are mapped so that basic block BB1 resides in shared instruction registers sir0 – sir16. Basic block BB2 resides in instruction registers ir17 – ir18 in processors EP0 – EP2. Basic block BB3 resides in instruction registers ir17 – 19 in processor EP3, and is aligned to basic block BB2. The alignment allows the jump as independent processors instruction at line 117 to correctly transfer control when the processors decouple and resume independent execution.

Figure 7.8 illustrates the instruction fetch and instruction issue schedules for each of the processors in the Ensemble. We can calculate the number of instruction pairs issued to the processors during the execution of one iteration of the loop as

$$4 \times 17 + 3 \times 2 + 3 = 77. \tag{7.3}$$

And we can calculate the number of instructions fetched from instruction registers during the execution of one iteration of the loop as

$$17 + 3 \times 2 + 3 = 26. \tag{7.4}$$

From 7.3 and 7.4, we can readily calculate that executing basic block BB1 in a processor corps reduces the number of instructions fetched from instruction registers by more than 66%.

7.4 Allocation and Scheduling

When applications with both data-parallel regions and thread-parallel regions are compiled, the compiler produces code that contains both shared instruction blocks and private instruction blocks. Control may transfer

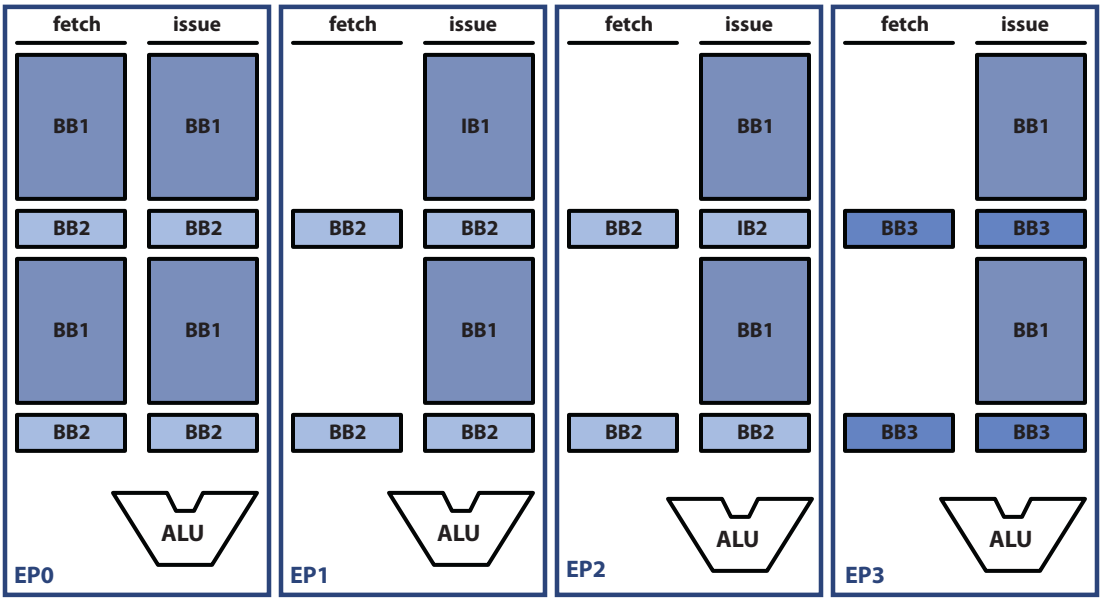


Figure 7.8 – Parallel fir Kernel Instruction Fetch and Issue Schedule.

frequently between the various shared and private instruction blocks. For example, code that contains vectorizable loops can be mapped to multiple processors by distributing loop iterations across the processors in an Ensemble. To improve instruction locality and reduce instruction register pressure, the compiler places the instructions from the vectorizable loops into shared instruction blocks, and places the remaining instructions into private instruction blocks. Similarly, data-parallel kernels that have been parallelized or replicated across the processors in an Ensemble exhibit both data-parallelism and thread-parallelism, and the resulting code may contain multiple shared and private instruction blocks that are finely interwoven in execution.

Figure 7.9 illustrates a basic scheme for allocating instruction registers in regions of code that contain both data-parallel and thread-parallel sections. The compiler partitions the instruction registers and uses partitions from each processor to form a shared instruction register pool. Registers in the shared pool are assigned to instruction blocks in data-parallel regions. The remaining instruction registers are considered private to each processor, and are assigned to instructions in thread-parallel regions, which are executed independently. The compiler allocates and schedules instruction registers using algorithms based on those described in Chapter 4, modified so that instructions in data-parallel blocks are assigned to instruction registers in the shared pool, which are managed cooperatively. Each processor is responsible for loading instruction blocks mapped to its private instruction register pool, and for loading shared instruction blocks mapped to its section of the shared pool.

Partitioning the instruction registers as illustrated in Figure 7.9 affords two significant benefits. First, it distributes the data-parallel instruction blocks across multiple processors. This allows multiple processors to load thread-parallel blocks in parallel, increasing the effective instruction register load bandwidth. Second, it provides a uniform private instruction register namespace. This allows the compiler to place thread-parallel

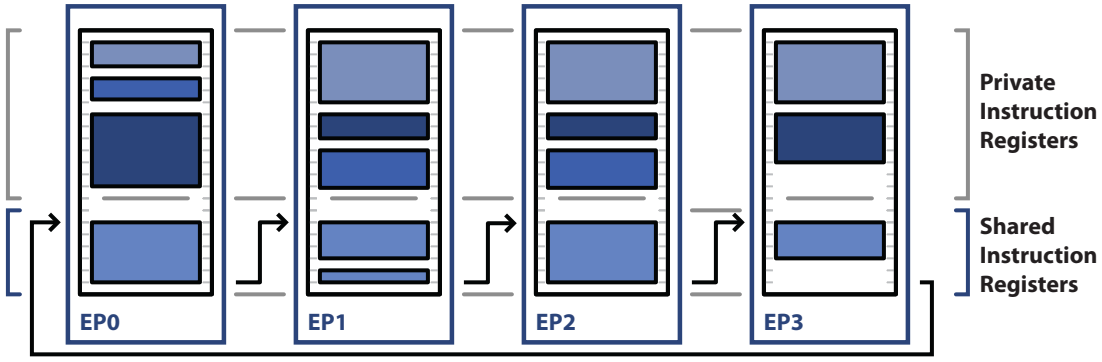


Figure 7.9 – Shared Instruction Register Allocation. The compiler partitions the instruction register files and combines aligned partitions from each processor to form a shared instruction register pool. The remaining instruction registers are private to each processor, and are available for instructions that are executed independently. Instruction blocks that are targets of control flow instructions that cause the processors to decouple are aligned in the private instruction register partitions. Illustrated as arrows in the figure, the compiler inserts jump instructions to transfer control between shared instruction blocks that reside in different processors.

instruction blocks at common addresses within each local private partition. As described subsequently, this simplifies code generation and improves performance.

When the instruction registers are partitioned as shown in Figure 7.9, the mapping of local instruction register identifiers to shared instruction register identifiers results in the private instruction registers partitioning the pool of shared instruction registers into 4 sets. The mapping of local identifiers to shared identifiers is described in detail in Appendix C. The shared registers within each processor reside at contiguous addresses, but the sets of private instruction registers are mapped to addresses between the shared register sets. The logical organization of the shared registers is illustrated in Figure 7.9. When necessary, the compiler constructs a contiguous pool of shared registers using control flow instructions to stitch together the registers in different register files. This allows the compiler to reason about the shared instruction register pool as a single contiguous block of registers. When an instruction block spans multiple register files, the compiler inserts explicit control flow instructions to transfer control between the shared registers in different instruction register files.

The compiler attempts to place instruction blocks so that those thread-parallel blocks that are successors of data-parallel blocks in the control flow graph are aligned at common locations in the private partitions. This allows control flow instructions that transfer control from data-parallel blocks to thread-parallel blocks to use a common destination register location to transfer control to blocks in the local private partitions. This allows the compiler to use control flow instructions that are resolved early in the pipeline to transfer control. When the compiler cannot align the destination blocks, the compiler inserts a register-indirect control transfer. Aligning the blocks improves performance and efficiency because control flow instructions that are resolved early in the pipeline impose smaller performance penalties.

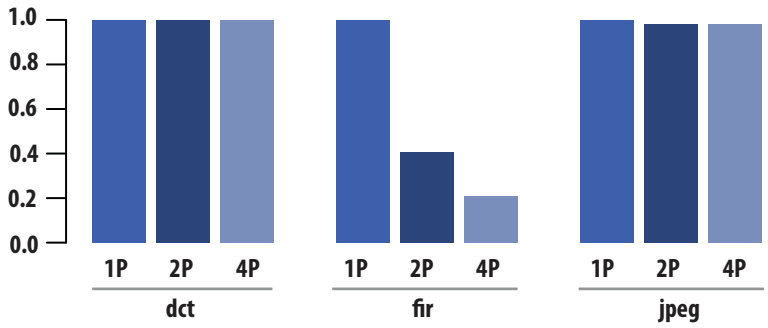


Figure 7.10 – Normalized Latency of Parallelized Kernels. Independent instances of the **dct** and **jpeg** kernels are mapped to different processors. The normalized latency of the **jpeg** kernel improves somewhat because the additional instruction registers eliminate instruction loads. The **fir** kernel is partitioned and mapped to multiple processors. Its normalized latency decreases because the additional registers allow more of the data working sets to be captured in registers, eliminating a significant fraction of all load instructions.

7.5 Evaluation and Analysis

This section presents an evaluation of the Ensemble mechanisms. We analyze the performance and efficiency impacts of mapping the **dct**, **fir**, and **jpeg** kernels to multiple processors within an Ensemble. The **dct** and **jpeg** kernels are parallelized by mapping independent instances of the kernels to multiple processors. The **fir** kernel is parallelized by partitioning the computation across multiple processors. The additional registers that are available to the compiler when the **fir** kernel is mapped to multiple processors allows more of its data working sets to be captured in registers.

Latency

Figure 7.10 shows the normalized latency of the parallelized kernels. The normalized latency measures the average time required to complete one invocation of the kernel. The normalized latency reported is measured by repeatedly invoking the kernels and dividing the aggregate execution time by the number of invocations. This amortizes the initial overheads associated with dispatching a kernel, such as fetching and loading instructions, across many invocations, and better reflects the steady state kernel latency. To illustrate the effects of mapping the kernels to multiple processors, the latencies are shown relative to the single processor implementations.

Throughput

Figure 7.11 shows the normalized throughput of the parallelized kernels. The normalized throughput is calculated by dividing the measured throughput by the number of processors. To illustrate the effects of mapping the kernels to multiple processors, the figure shows the normalized throughput relative to the single processor implementations. Mapping a kernel to multiple processors increases the computation and storage resources

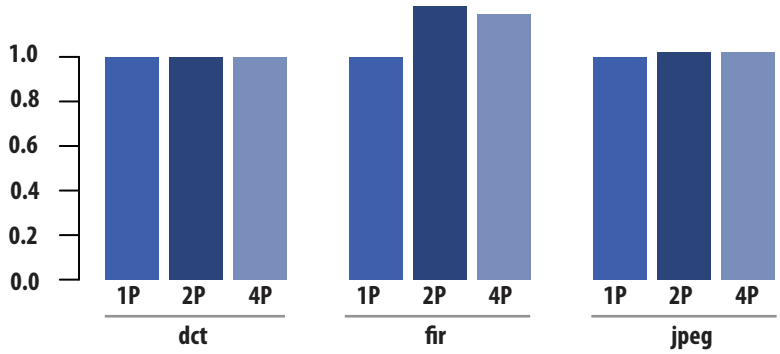


Figure 7.11 – Normalized Throughput of Parallelized Kernels. The normalized throughput adjusts for the number of processors to provide a measure of area efficiency.

that are assigned to the kernel, and the normalized throughput provides a measure of area efficiency by adjusting for the number of processors the computation is distributed across.

Mapping the **dct** kernel to multiple processors does not affect area efficiency. The kernel iterations are applied in parallel and aggregate throughput increases, but the throughput per processor remains constant. Mapping the **fir** kernel to multiple processors improves efficiency by eliminating memory accesses. However, the four-processor configuration is marginally less area efficient because of the additional communication and synchronization instructions at the end of the kernel, the distribution of which leads to minor load imbalance. The **jpeg** kernel exhibits some efficiency improvement because the additional instruction capacity provided by the shared instruction registers reduce the number of instructions that are loaded. The observed improvement is data dependent because the entropy encoding time depends on the data to be encoded and may vary significantly. This introduces load imbalance and the potential for poor processor utilization. Furthermore, the DC coefficient is encoded differentially across blocks, which introduces a data dependency that is carried across kernel iterations. We should note that many modern image compression standards provide coding modes that allow parts of an image to be compressed and decoded in parallel, typically by allowing the image to be partitioned into stripes or tiles that are encoded and decoded independently. The coded bit-stream is constructed by concatenating the compressed streams generated for the individual tiles or slices.

Instruction Efficiency

Figure 7.12 compares the aggregate energy consumed delivering instructions to the processors when the kernels are mapped to multiple processors. The aggregate energy includes the energy consumed fetching instructions from instruction registers and transferring instructions to instruction registers from the backing memory. The data shown in Figure 7.12 reflects the energy consumed by all of the processors during one kernel iteration, and does not account for differences in the amount of computation performed when kernels are mapped to multiple processors. For example, when mapped to one processor, the **dct** kernel computes a single transform during each iteration; when mapped to four processors, it computes four transforms during each iteration, and therefore the useful computation performed is four times greater. Consequently, it is difficult

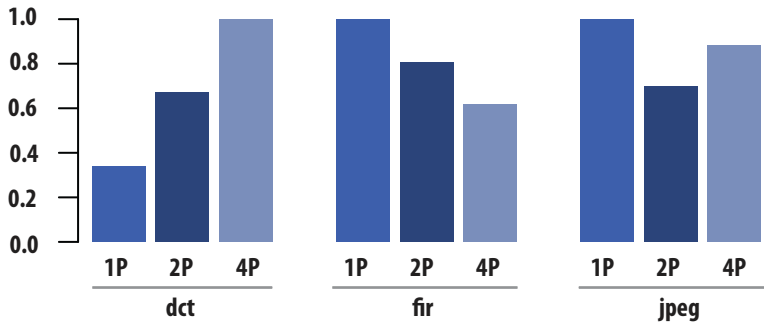


Figure 7.12 – **Aggregate Instruction Energy.** The aggregate instruction energy measures the total energy consumed delivering instructions to the processors. The data have been normalized within each kernel to illustrate trends.

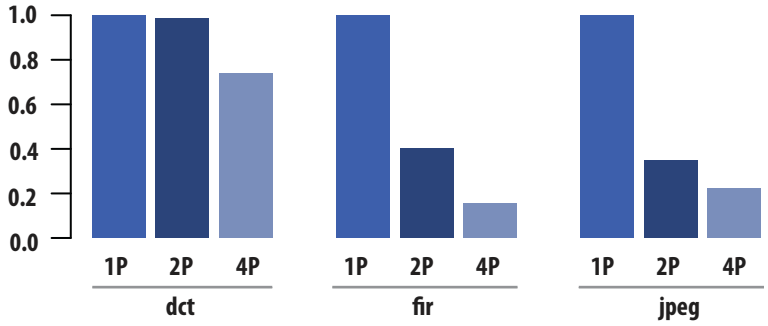


Figure 7.13 – **Normalized Instruction Energy.** The normalized instruction measures the ratio of the energy consumed delivering instructions to the number of processors, and provides a measure of energy efficiency.

to infer too much from Figure 7.12 about the efficiency of using processor corps to map kernels to multiple processors. However, the data shown in Figure 7.12 provides insights into how efficiency is improved and exposes limitations. The data for the **dct** kernel show that energy consumed distributing instructions to multiple processors is comparable to the energy consumed accessing instructions stored in local instruction registers. This illustrates how the effectiveness of using SIMD execution to amortize instruction supply energy is limited by the cost of distributing instructions, and how inexpensive local instruction stores reduce some of the efficiency advantages of SIMD execution. The data also illustrate that the significant improvement in the efficiency of the **fir** kernel results because fewer instructions are executed when the kernel is mapped to multiple processors; the reduction in the number of instructions executed is more than sufficient to compensate for the additional energy consumed distributing instructions to multiple processors. Figure 7.13 shows the normalized instruction energy, calculated by dividing the aggregate instruction energy by the amount of useful computation completed, which provides a direct measure of energy efficiency.

The **dct** kernel exposes limits to the improvement in efficiency that are achieved by distributing instructions to multiple processors. The aggregate instruction energy of the **dct** kernel is dominated by the fetching of instructions from instruction registers and the distributing of them among the processors comprising a

processor corps. Each processor has enough instruction registers to capture both kernels entirely, so the additional instruction register capacity provided by the shared instruction register pool does not reduce the number of instructions that are transferred to the instruction register files. Furthermore, the initial cost of loading instructions when the kernels is launched is negligible when amortized across many kernel iterations.

The increase in the aggregate instruction energy observed when the **dct** kernel is mapped to more processors exposes the expense of transferring instructions to remote processors. Interconnect capacitance increases with the physical distance instructions traverse, and the energy expended distributing instructions increases with the capacitance that is switched when instructions are delivered to remote processors. Thus, though the number of instruction register accesses remains constant, the loss of physical locality increases the cost of delivering instructions. As Figure 7.12 illustrates, even local communication can require a significant expenditure of energy, and improvements in energy efficiency are limited by the loss of physical locality. However, less energy is required to transfer instructions to remote processors than is required for additional processors to fetch instructions from their local instruction registers, and modest efficiency improves are realized when the kernel is mapped to a processor corps, as Figure 7.13 shows.

The effectiveness of exchanging instruction locality for fewer instruction store accesses depends on the relative cost of distributing instructions and accessing local instructions. These costs can vary significantly across different architectures. For example, accessing an instruction cache consumes more energy than accessing an instruction register file. Consequently, processors that fetch instructions from instruction caches will benefit more from architectures that allow instructions to be issued to multiple processors.

Like the **dct** kernel, the aggregate instruction energy of the **fir** kernel is dominated by the fetching of instructions from instruction registers and the distributing of them among the processors comprising a processor corps. There are enough instruction registers to accommodate the kernel, and the initial cost of loading instructions when the kernel is launched is negligible when amortized across many kernel iterations. The normalized instruction energy for the **fir** decreases because fewer instructions are executed. The additional indexed registers accommodate more of the working sets, and fewer memory operations are performed.

The normalized instruction energy for the **jpeg** kernel is more informative because the instruction registers lack sufficient capacity to capture instruction reuse across subsequent invocations of the kernel. The DCT transform component is performed in parallel, and the instructions in the shared instruction block are read once, amortizing the memory access across multiple processors. The Huffman encoding component is executed by the processors independently. The initial instruction block is loaded and distributed to all of the processors, after which the instruction loads diverge as the processors follow different control paths. In contrast with the **dct** kernel, which does not load instructions during each iteration, the **jpeg** kernel demonstrates both the effectiveness of amortizing instruction issue as well as instruction loads by executing code.

7.6 Related Work

The INMOS transputer architecture [149] allows machines of different scales to be constructed as collections of many small, simple processing elements connected by point-to-point links. The architecture reflects a decentralized model of computation in which concurrent processes operate exclusively on local data and

communicate by passing messages on point-to-point channels. The transputers implement a stack-based load-store architecture that is capable of executing a small number of instructions. Like the HEP multiprocessor described subsequently, each processing element can execute multiple processes concurrently. Channels that connect processes executing on the same transputer are implemented as single words in memory. Channels that connect processes executing on different transputers are implemented by point-to-point links. In this respect, the programming model mirrors the physical machine design. The restriction that all communication be mapped to point-to-point channels complicates the task of mapping applications to a transputer. Early transputer implementations used a synchronous message passing protocol that required that both the sending process and receiving process are ready to communicate before a message can be transferred, which limits the machine's ability to overlap computation and communication.

The HEP multiprocessor system uses multithreading to hide memory latency and allows cooperating concurrent threads to communicate through shared registers and memory [134]. The system supports fine-grained communication and synchronization by associating an access state with every register memory and data memory location to record whether a word is full, empty, or reserved. Load instructions can be deferred until the addressed data word is full, and can atomically load the data and mark the location as empty. Store instructions can be deferred until the addressed word is empty, and can atomically store the data to the addressed memory location and mark the location as full. An instruction that accesses register memory can be deferred until all of its operand register locations are marked full and its destination register location is marked empty; when the instruction executes, it atomically marks its operand registers as empty and its destination register as full. To enforce atomicity, the destination register location is temporarily marked as reserved during the execution of the instruction, and other instructions may not execute if any of their registers are marked reserved. Multiprocessor data flow architectures use similar full-empty bit mechanisms to detect when the operands to an operation have been computed [113], which is fundamental to implementing the underlying model of computation on data flow machines [9].

The MIT Alewife machine implements full-empty bit synchronization using a similar scheme to improve support for fine-grain parallel computation and communication through shared memory [4]. The architecture associates a synchronization bit with each memory word. The synchronization bits are accessed through distinguished load and store operations that perform a test-and-set operation in parallel with the memory access. The architecture allows memory operations to signal a synchronization exception when the test-and-set operation fails [3]. The original state of the synchronization bit is returned in the coprocessor status word when the memory operation completes, and special branch instructions are implemented to allow software to examine the synchronization bit.

The MIT Raw microprocessor [137] exposes a scalar operand network [138] that allows software to explicitly route operands between distributed function units and register files. The distributed function units and register files in Raw are organized as scalar processors, and Raw's scalar operand network effectively allows software to compose and receive messages in registers to reduce the latency of sending and receiving messages comprising single scalar operands. This allows instructions from a single thread to be distributed across multiple execution units. Scalar operand networks provide a general abstraction for exposing interconnect to software, with the intention that exposing interconnect delays and allowing software to explicitly

manage interconnect resources will allow software to schedule operations to hide communication latencies between distributed function units and register files.

Vector-thread architectures allow independent threads to execute on the vector lanes [88]. This allows data parallel codes that are not vectorizable to be mapped efficiently to the virtual processors of a vector-thread machine, and allows the vector lanes to be used to execute arbitrary independent threads. Vector lane threading allows independent threads to execute on the lanes of a multi-lane vector processor [120]. This improves lane utilization during applications phases that exhibit insufficient data-level parallelism to fully utilize the parallel vector lanes. By decoupling hardware intended for single-instruction multiple-data execution, both vector-thread processors and vector processors that support vector-lane threading are able to use the same hardware to exploit both single-instruction multiple-data parallelism and thread-level parallelism.

The vector-thread abstract machine model connects virtual processors using a unidirectional scalar operand ring network [88]. The network allows an instruction that executes on virtual processor n to transfer data directly to an instruction that executes on virtual processor $n+1$. Though communication between virtual processors is restricted, these cross virtual processor transfers allow loops with cross-iteration dependencies to be mapped onto the vector lanes. The execution cluster organization used in the Scale vector-thread processor uses queues to decouple the send and receive operations that execute on different clusters [89].

The Multiflow Trace/500 machine implements multiple function unit clusters and uses distributed local instruction caches [24]. Machines may be configured with two coupled VLIW processors, each with 14 function units, that execute a common instruction stream. The two processors appear to software as a single VLIW processor with twice the number of function units.

The variable-instruction multiple-data (XIMD) architecture [151] extends conventional very-long instruction word (VLIW) architectures [42] by associating independent instruction sequencers with each function unit, which allows a XIMD processor to execute multiple instruction streams in parallel. Distributed condition code bits and synchronization bits are used to control the instruction sequencers. The sequencers and distributed synchronization bits allow a XIMD machine to implement barrier synchronization operations that support medium-grain parallelism. The independent sequences and mechanisms that support inter-function unit synchronization allow for some control flow dependencies to be resolved dynamically. This allows a XIMD machine to emulate a tightly-coupled multiple-instruction multiple-data (MIMD) machine. As with traditional VLIW architectures, function units communicate through a global register file. Like VLIW machines, XIMD machines require sophisticated compilers to identify and exploit instruction-level parallelism. In addition, many of the architectural mechanisms that distinguish XIMD machines require sophisticated compiler technology to distribute and schedule instructions from multiple loop iterations across the distributed function units [111].

The distributed VLIW (DVLIW) architecture [155] extends a clustered VLIW architecture with distributed instruction fetch and decode logic. Each cluster has its own program counter and local distributed instruction cache, which enables distributed instruction sequencing. Operations that execute on different clusters are stored in the distributed instruction caches, similar to how instructions are stored in early horizontally microcoded machines [142]. Later extensions proposed coupling processors to support both instruction-level and thread-level parallelism [156]. The resulting architecture is similar to the MIT Raw microprocessor,

except that it uses a few large VLIW cores rather than many small RISC cores. As with Raw, cores are connected by a scalar operand network. The network is accessed through dedicated communication units that are exposed to software. The instruction set supports both direct scalar operand puts and gets between adjacent cores, and decoupled sends and receives between all cores. Data transferred using sends and receives traverse inter-cluster decoupling queues. The compiler statically schedules instructions to exploit instruction-level parallelism within a core. A single thread may be scheduled across multiple cores using gets and puts to synthesize inter-cluster moves in a traditional clustered VLIW machine, or multiple threads may be scheduled across the cores. The architecture requires a memory system that supports transactional coherence and consistency [62] to dynamically resolve memory dependencies, which significantly complicates the design and implementation of the memory system.

7.7 Chapter Summary

Ensembles allow software to use collections of coupled processors to exploit data-level and task-level parallelism efficiently. The Ensemble organization supports the concept of processor corps, collections of processors that execute a common instruction stream. This allows processors to pool their instruction registers, and thereby increase the aggregate register capacity that is available to the corps. The instruction register organization improves the effectiveness of the processor corps concept by allowing software to control the placement of shared and private instructions throughout the Ensemble.

Chapter 8

The Elm Memory System

This chapter introduces the Elm memory system. Elm exposes a hierarchical and distributed on-chip memory organization to software, and allows software to manage data placement and orchestrate communication. Additional details on the instruction set architecture appear in Appendix C. The chapter is organized as follows. I first explain general concepts and present the insights that informed the design of the memory system. I then describe aspects of the microarchitecture in detail. This is followed by an example and an evaluation of several of the mechanisms implemented in the memory system. Finally, I consider related work and conclude.

8.1 Concepts

Elm is designed to support many concurrent threads executing across an extensible fabric of processors. To improve performance and efficiency, Elm implements a hierarchical and distributed on-chip memory organization in which the local Ensemble memories are backed by a collection of distributed memory tiles. The architectural organization of the Elm memory system is illustrated in Figure 8.1. The organization allows the memory system to support a large number of concurrent memory operations. Elm exposes the memory hierarchy to software, and lets software control the placement and orchestrate the communication of data explicitly. This allows software to exploit the abundant instruction and data reuse and locality present in embedded applications. Elm lets software transfer data directly between Ensembles using Ensemble memories at the sender and receiver to stage data. This allows software to be mapped to the system so that the majority of memory references are satisfied by the local Ensemble memories, which keeps a significant fraction of the memory bandwidth local to the Ensembles. Elm provides various mechanisms that assist software in managing reuse and locality, and assist in scheduling and orchestrating communication. This also allows software to improve performance and efficiency by scheduling explicit data and instruction movement in parallel with computation.

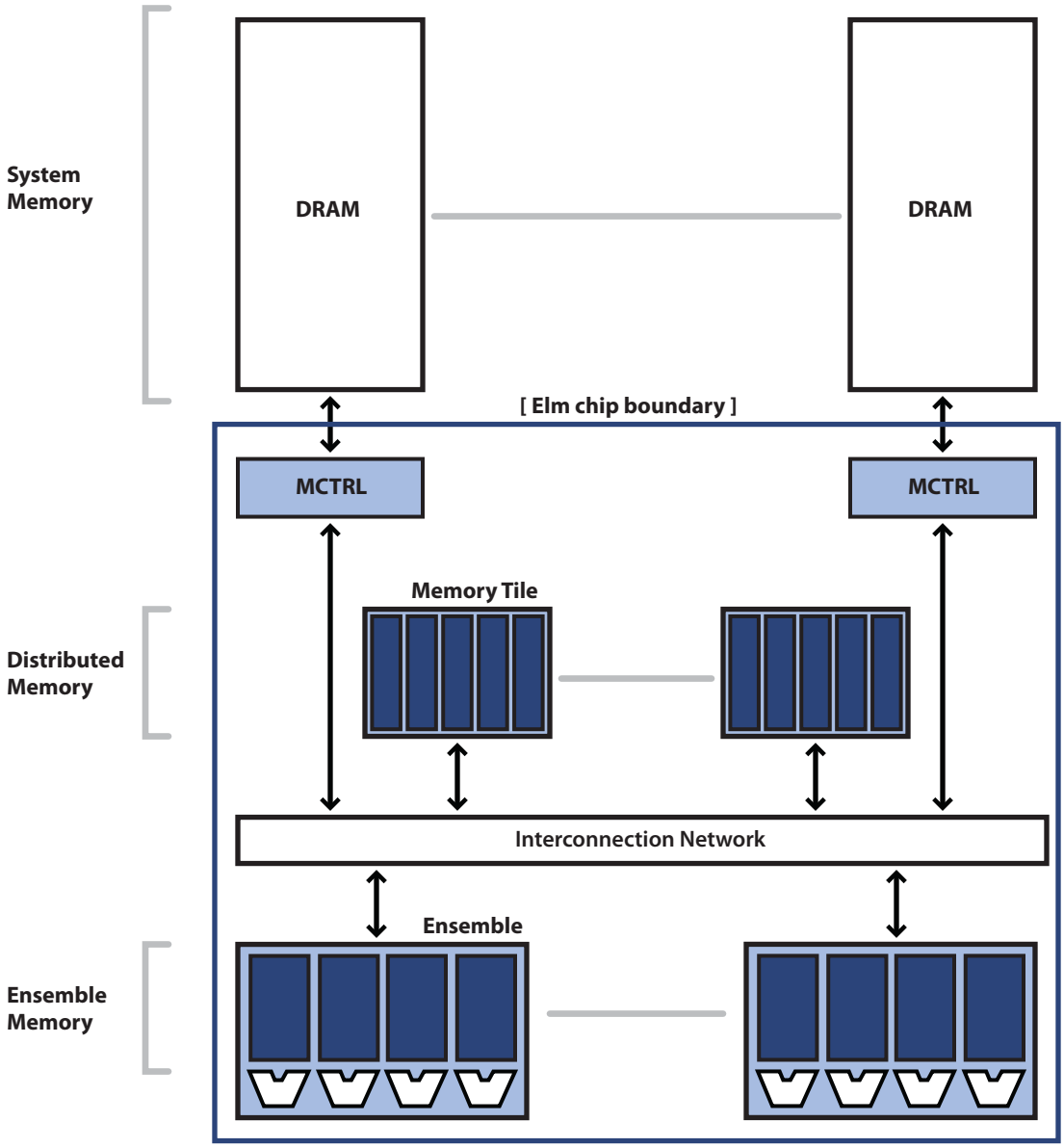


Figure 8.1 – **Elm Memory Hierarchy.** The memory hierarchy is composed of Ensemble memories, distributed memories, and off-chip system memory. The system memory is accessed through memory controllers (MCTRL). The Ensemble memories are used to stage instruction and data transfers to Ensemble processors, and to capture small working sets locally. The distributed memories are used to capture larger instruction and data working sets, to stage instruction and data transfers up and down the memory hierarchy, and to stage data transfers between Ensembles.

Parallelism and Locality

Elm provides a global shared address space and implements a distributed shared memory system. Applications are partitioned and mapped to threads that execute concurrently on different processors. Communication through shared memory is implicit, and the global shared address space supports dynamic communication patterns, where matching two-sided messages would otherwise be inconvenient. The shared address space allows software to create explicit shared structures that are accessed through pointers. As illustrated in Figure 8.2, the mapping of addresses onto physical memory locations exposes the hierarchical and distributed memory organization. The partitioning of the address space allows software to control the placement of instructions and data explicitly. It also provides software with a notion of proximity that allows it to exploit locality by mapping instructions and data to memory locations that are close to the processors that access them. This allows software to exploit locality by placing instructions and data close to the processors that require them, and to schedule computation close to resident data. Elm allows software to orchestrate the horizontal and vertical movement of instructions and data throughout the memory system. The memory system provides abstract block and stream operations to optimize explicit bulk communication.

The distributed and hierarchical memory organization allows a significant fraction of memory references to be mapped to local memories. This improves efficiency by reducing demand for expensive remote memory and global interconnection network bandwidth. The distributed organization allows the memory system to handle many concurrent access streams, and allows software to keep shared instructions and data sets close to collections of collaborating processors.

The Ensemble memories are used to extend the capacity of the local register files and to stage data and instruction transfers to the processors within an Ensemble. The compiler typically reserves part of each Ensemble memory to extend the capacity of the local instruction and data registers. Essentially, the compiler treats these locations as additional register locations when allocating instruction and data registers, and uses them to capture instruction and private data that is temporarily spilled to memory. Shared regions of an Ensemble memory can be used to transfer data directly between processors without moving the data through distributed and system memory.

The distributed memory tiles provide memory capacity to decouple instruction and data transfers, to capture large instruction and data working sets, and to capture shared working sets. The memories are physically distributed across the chip to allow instructions and data to be captured close to the processors that access them. This also allows communication to be mapped to memories that are physically close to the communicating processors to reduce the energy expended transporting the data, and to reduce the demand for global interconnection network bandwidth. Software uses the distributed memories to stage data and instruction transfers between system memory and processors. Shared instructions and data can be transferred to a distributed memory before being forwarded to multiple Ensembles. This reduces off-chip memory accesses, which improves energy efficiency, and reduces demand for system memory bandwidth.

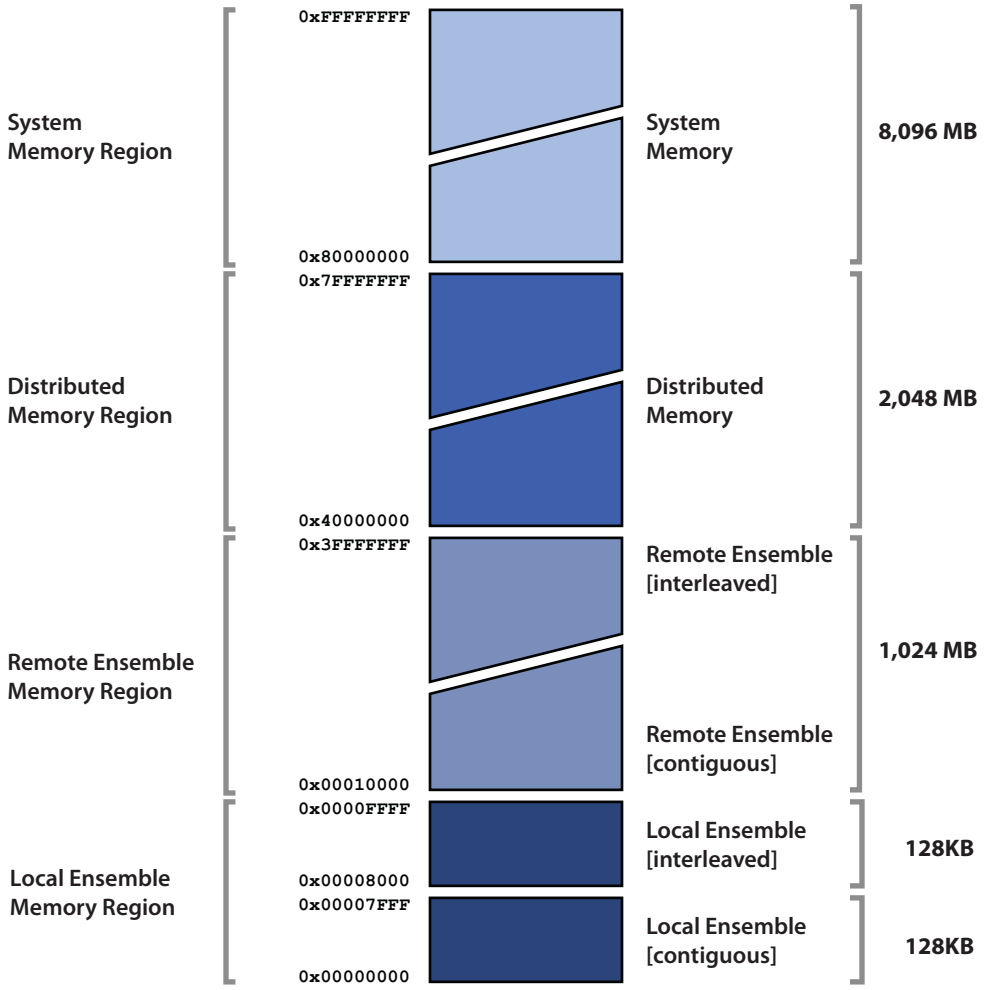


Figure 8.2 – **Elm Address Space.** The capacity shown for each region reflects the partitioning of the address space rather than the capacity of the physical memories. To simplify the construction of position-independent code, the local Ensemble memory is always mapped at a fixed position in the address space.

Distributed Memory and Caching

Distributed memory provides additional shared storage capacity close to the Ensembles. Distributed memory serves three important purposes: it is used to stage instruction transfers from system memory to Ensemble memory and instruction registers; it is used to stage data transfers between system memory and Ensemble memory; and, it is used to stage data transfers between Ensembles. The additional storage capacity provided by distributed memory allows software to use techniques such as prefetching to tolerate long memory access times.

The distributed memory tiles are nominal software-managed memory. However, Elm allows distributed memory regions to operate as hardware-managed caches. The caches may be used to filter references to

system, distributed, and remote Ensemble memory. Software-managed distributed memory allows software to coordinate reuse and locality, and to orchestrate the movement of data through the memory hierarchy explicitly. This improves efficiency and predictability when mapping software that is amenable to static analysis. Caches assist software in managing reuse and locality when access patterns and data placement are difficult or impossible to determine using static analyses. The caches can also be used to filter references to shared objects that are mapped to system memory, which keeps more memory bandwidth closer to the processors. This improves efficiency by keeping more communication close to the processors and reducing accesses to more expensive system and remote memory.

Caches soften the process of mapping applications to Elm. The caches exploit locality using conventional load and store memory operations. They also provide a reasonable way to capture instruction blocks close to processors without the complexity of using Ensemble memory to store both instructions and data.

Elm does not implement hardware cache coherence. Instead, software may be constructed to ensure that any cached copies of shared data that may be updated are cached at a common location on the chip, or may explicitly invalidate cached copies of shared data. Elm provides efficient synchronization mechanisms that software may use to coordinate the invalidating of cached copies of shared data.

Software-Constructed Cache Hierarchies

The caches in Elm are software-allocated and hardware-managed. Software designates parts of the distributed memory tiles to operate as caches, and determines which regions of memory are accessed through the caches. Elm allows software to control the mappings from cached addresses to the distributed memory tiles that cache the addresses. This allows software to construct cache hierarchies, which affords software significant control over where instructions and data are cached, and when multiple cached copies of instructions and data may exist. The flexibility lets software map overlapping working sets to shared caches, which improves cache utilization and reduces memory bandwidth demand.

Elm associates a small cached address translation table with each processor and each distributed memory tile to record the mappings from cached addresses to cache controllers. Each entry describes a contiguous region of the address space and associates a distributed memory tile or system memory controller with the region. The mapping may restrict the ways of the cache that are used to cache the data, which effectively allows software to partition cache resources within a memory tile. When a memory controller handles a cached memory operation, it uses its local cached address translation tables to determine where the operation should be forwarded to when a request cannot be handled locally. The tables are small, and typically provide 4 entries.

The cached address translation tables provide a simple mechanism that allows software to partition cache capacity and to isolate access streams. For example, software may use one distributed memory tile to cache addresses that contain instructions, and a different memory tile to cache addresses that contain data. This isolates the instruction and data streams and keeps them from displacing each other in the caches. To improve locality, software may configure the translation tables so that distant processors use local tiles to distribute cached copies of shared instructions and read-only data. Similarly, to distribute memory references and

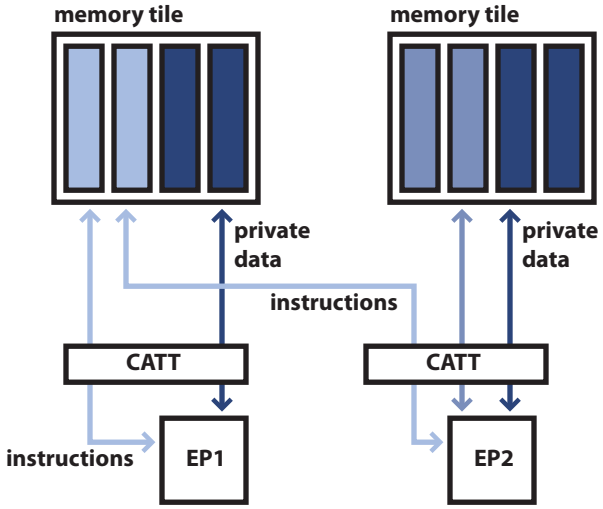


Figure 8.3 – **Flexible Cache Hierarchy.** The cached address translation table (CATT) associated with each processor allows software to control which distributed memory tiles are used to cache the instructions and data accessed by different processors. This allows software to capture overlapping working sets in a shared cache, and to distribute private data across multiple caches.

bandwidth demand across multiple tiles, software may configure the translation tables to use multiple tile to cache shared data.

Decoupled Memory Operations

Memory operations that access remote distributed and system memory may require 10s to 100s of cycles to complete. In certain cases, data and control dependencies in an application require that a processor wait for a memory operation to complete before executing subsequent instructions. However, it is often possible to overlap memory operations with independent computation.

Elm processors complete conventional load and store operations in order. After injecting the operation into the memory system, the processor waits for the memory system to indicate that it has completed the operation before executing subsequent instructions. Consequently, long memory access times cause performance to degrade when software accesses memory beyond the local Ensemble.

Elm provides decoupled load and store operations that allow a processor to continue to execute independent instructions while the memory system handles memory operations. Decoupling the initiation and completion of memory operations improves performance when accessing remote memory by allowing software to overlap independent computation with the memory access time. It also allows software to tolerate better unpredictable memory access times, for example when competing for access to the Ensemble memory. Decoupled memory operations let software exploit the available parallelism and concurrency in the memory system.

Decoupled loads are injected into the memory system when the load operation executes, and the destination register is updated when the memory system returns the load data. Hardware tracks pending memory operations, and the pipeline interlock logic stalls the pipeline when it detects a read-after-write hazard involving a pending load operation. A processor may have multiple decoupled loads pending. Decoupled stores are injected into the memory system when the store operation executes, and subsequent memory operations are allowed to proceed before the processor receives a message indicating the store has been committed.

The Elm memory system does not provide strong memory ordering, and decoupled memory operations may complete in a different order from that in which they are issued by a processor. However, conventional load and store operations are guaranteed to complete in program order because at most one may be in flight, and consequently may be used when software requires strong memory ordering. The instruction set provides an instruction that causes execution to suspend until all pending memory operations have completed, which effectively implements a memory fence.

Stream Memory Operations

Stream memory operations provide an efficient mechanism for moving blocks of instructions and data through the memory system. Stream loads and stores operate on regular sequences of memory blocks. Elm uses stream descriptors to describe the composition of streams. Stream descriptors effectively provide a compact representation of the addresses comprising a stream. Stream memory operations support non-unit stride memory accesses, and allow sequences to be gathered and scattered. Examples of the different addressing modes are illustrated in Figure 8.4 through Figure 8.7. Streams often correspond to sequences of records in applications, and stream memory operations provide an efficient mechanism for software to assemble and distribute collections of records. Stream operations allow hardware to efficiently convert data between array-of-structures and structures-of-arrays representations when moving data through the memory system.

Stream controllers distributed throughout the memory system perform stream memory operations. The stream controllers contain small collections of stream registers, which are used to store active stream descriptors. Each stream register contains a stream descriptor register and a stream state register. The stream state register records the state of stream memory transfer. Typically, software loads a stream descriptor into a stream register and then issues a sequence of stream memory operations that reference the stream descriptor. Each stream memory operation specifies how many stream elements should be transferred, which may differ from the number of elements in the stream.

Decoupling the specification and transfer of the stream descriptor from the stream memory operations provides several advantages. It allows large transfers to be decomposed into multiple smaller transfers without imposing the overhead of repeatedly transferring the stream descriptor to the stream engine. It also allows multiple clients to share a stream descriptor and use hardware to serialize and interleave concurrent accesses. Hardware completes one stream memory transaction at a time, and therefore serializes stream memory operations. For example, a stream descriptor may be configured to describe the elements of a circular buffer, and stream memory operations maybe be used to append and remove blocks from a queue implemented using the circular buffer. Additional coordination variables are needed to coordinate append and remove operations to

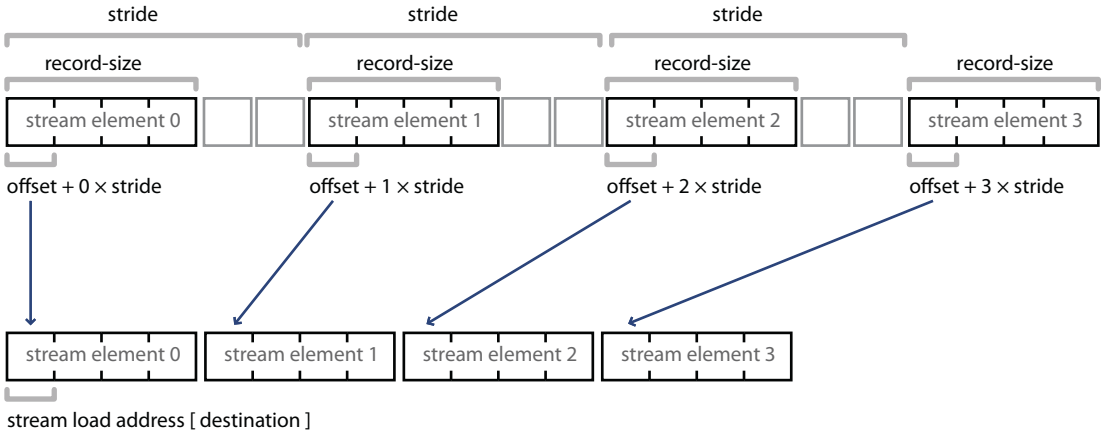


Figure 8.4 – **Stream Load using Non-Unit Stride Addressing.** The record addresses are computed by adding multiples of the stride to the offset address specified for the stream operation. The record size, stride, and number of records in the stream are encoded in a stream descriptor; the offset address, source addresses, and number of records transferred are specified when the operation is initiated.

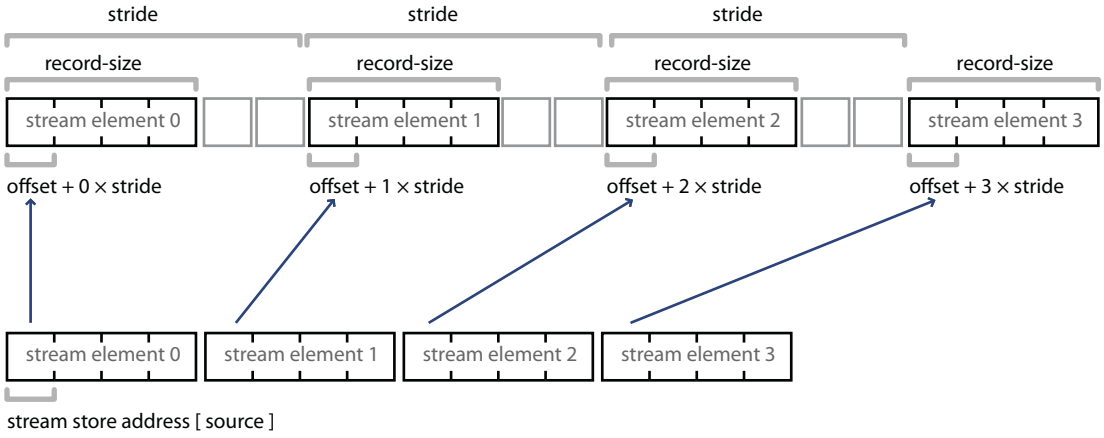


Figure 8.5 – **Stream Store using Non-Unit Stride Addressing.** The record addresses are computed by adding multiples of the stride to the offset address specified for the stream operation. The record size, stride, and number of records in the stream are encoded in a stream descriptor; the offset address, source addresses, and number of records transferred are specified when the operation is initiated.

ensure enough elements are in the buffer, but concurrent append and remove operations can be implemented using stream memory operations.

Stream memory operations may use memory tiles as caches. Because stream memory operations are decoupled from execution, caching is used to filter references to off-chip memory and reduce remote memory bandwidth demand rather than to improve latency.

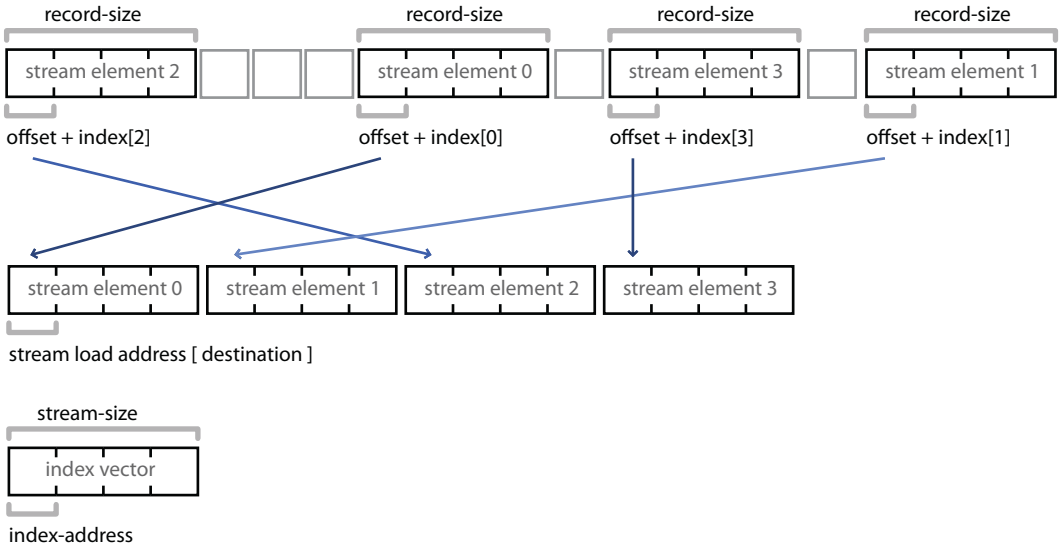


Figure 8.6 – **Stream Load using Indexed Addressing.** The record addresses are computed by adding offsets specified by an index vector to the offset address specified for the stream operation. The record size, address of the index vector, and number of records in the stream are encoded in a stream descriptor; the offset address, source addresses, and number of records transferred are specified when the operation is initiated.

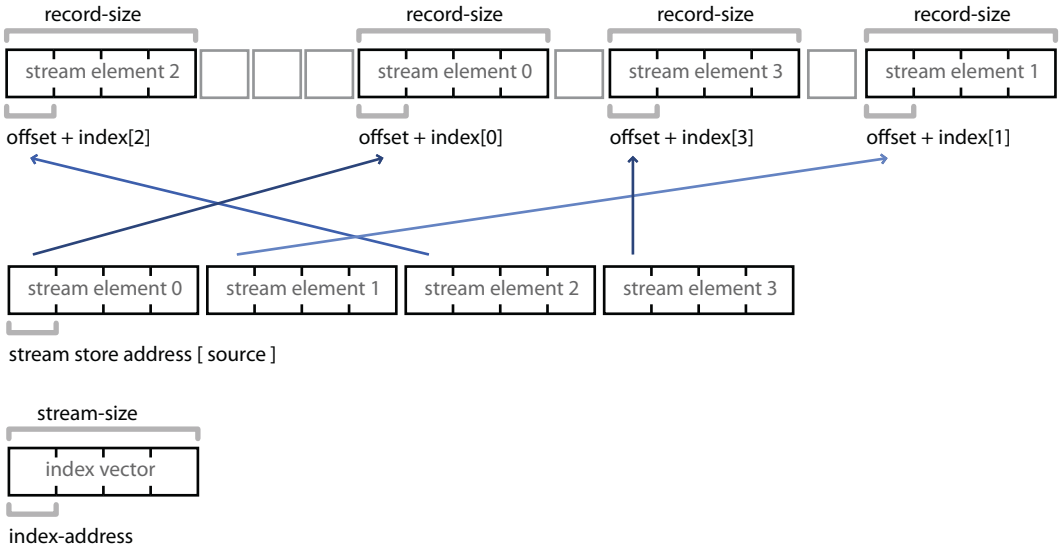


Figure 8.7 – **Stream Store using Indexed Addressing.** The record addresses are computed by adding offsets specified by an index vector to the offset address specified for the stream operation. The record size, address of the index vector, and number of records in the stream are encoded in a stream descriptor; the offset address, source addresses, and number of records transferred are specified when the operation is initiated.

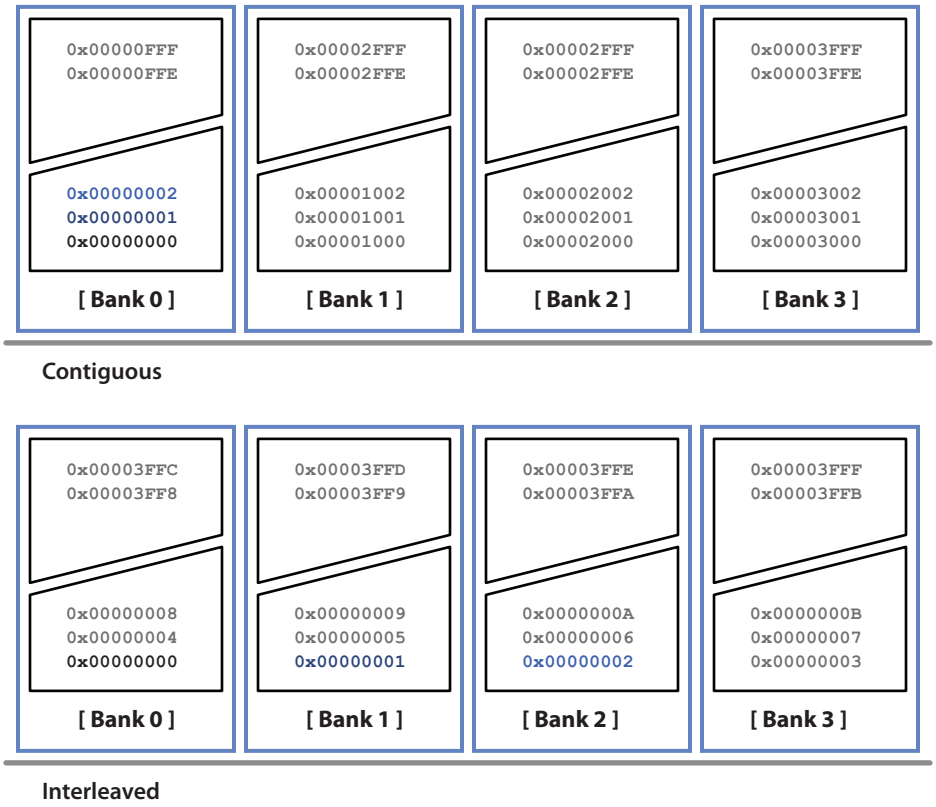


Figure 8.8 – **Local Ensemble Memory Interleaving.** Memory operations that transfer data between a contiguous region of memory and an interleaved region of memory can be used to implement conversions between structure-of-array and array-of-structure data object layouts.

Memory Interleaving

Elm maps each physical Ensemble and distributed memory location into the global address space twice: once with the memory banks contiguous, and once with the memory banks interleaved. In the contiguous address range, the capacity of the Ensemble memories determines which address bits are used to map addresses to memory banks. In the local interleaved Ensemble address range, the least significant address bits select the bank. The difference is illustrated in Figure 8.8. The contiguous and interleaved address ranges map to the same physical memory, so each physical memory word is addressed by two different local addresses.

The contiguous Ensemble addressing scheme is useful when independent threads are mapped to an Ensemble. A thread may be assigned one of the local memory banks, and any data it accesses may be stored in its assigned memory bank. This avoids bank conflicts when threads within an Ensemble attempt to access the local memory concurrently, which improves performance and allows for more predictable execution. Because consecutive addresses map to the same memory bank, all of the data accessed by the thread resides within a contiguous address range, and no special addressing is required to avoid bank conflicts.

The interleaved Ensemble addressing scheme is useful when collaborative threads are mapped to an Ensemble. Adjacent word addresses are mapped to different banks, and accesses to consecutive addresses are distributed across the banks. This allows data transfers between local Ensemble memory and remote memory to use consecutive addresses while distributing the data across multiple banks, which improves bandwidth utilization when the memories are accessed by the processors within the Ensemble.

The mapping of addresses to Ensemble memory banks in the remote Ensemble address range can be configured by software. This allows software to control the granularity of bank interleaving. Configuration registers in the memory controllers let software configure the logical interleaving of memory banks by selecting which address bits determine the mapping of addresses to banks. Stream memory operations can be used when software requires greater control over the mappings of local addresses to remote addresses, and when different accesses prefer different mappings.

Coordination and Synchronization

An Elm application is typically partitioned into multiple kernels that are mapped onto threads executing across multiple processors. In addition to communicating through shared memory locations, these threads may communicate by passing messages through the local communication fabric. Regular local communication patterns can be effectively mapped to the local communication fabric, which provides an efficient communication substrate that combines communication and synchronization. Shared memory provides a more general communication substrate, but requires explicit synchronization. Because the local communication fabric provides efficient support for fine-grain communication and synchronization, the mechanisms implemented to support communication through shared memory are not optimized specifically for efficient fine-grain communication. Instead, they are designed to support a large number of concurrent and potentially independent communication and synchronization flows.

Shared memory that is not accessed through caches can always be used to implement coordination variables [63]. Each memory controller provides a synchronization points that establishes an order on memory operations that access shared memory managed the controller. Coordination objects may be mapped to memory locations that are close to the processors that access them to improve locality and reduce memory access latencies.

Shared memory that is accessed through a cache may be used to implement coordination variables provided that all accesses to a shared data object access the data through a shared cache hierarchy. This restriction ensures that all processors observe the current state of the shared objects. Coordination objects located in system memory may be accessed through a cache to reduce memory access times. Because the Elm memory system does not enforce cache coherence in hardware, software must ensure that copies of coordination objects are not cached in multiple locations. Software may explicitly force the release of cached data, which allows copies of shared data that may reside in caches to be returned to memory at the end of specific application phases.

Elm provides hardware synchronization primitives that let software efficiently implement more sophisticated coordination objects and concurrent data structures. Elm provides an atomic compare-and-swap instruction and an atomic fetch-and-add instruction, which are two relatively common atomic read-modify-write memory operations [90]. These operations may be performed on cached shared memory locations provided that all accesses to the shared objects access a shared cache hierarchy; this ensures that all accesses see the current state of the shared objects.

The atomic compare-and-swap operation accepts three parameters: a memory address, an expected value, and an update value. The operation returns the current value stored at the memory location referenced by the address parameter. If the current value matches the expected value, the update value is stored to the memory location. The atomic compare-and-swap operation is provided to support arbitrary parallel algorithms. Memory locations that implement atomic compare-and-swap operations have an infinite consensus number [61], and a wait-free implementation of any concurrent data object can be implemented using compare-and-swap operations.

The atomic fetch-and-add operation has a consensus number of exactly two [2], which limits its efficacy as a means of constructing general wait-free concurrent data objects [63]. However, it can be used to construct efficient implementations of many common and important operations on shared data objects found in embedded applications. For example, it allows software to implement efficient strategies for allowing multiple threads to concurrently insert and remove data objects from shared queues, and it allows updates such as those used to compute histograms to proceed concurrently.

The atomic fetch-and-add instruction also accepts three parameters: a memory address, an update value, and sequence value. The operation returns the current value stored at the memory location referenced by the address parameter. The value stored in the memory location is replaced with the sum of the current value and update value except when the sum is equal to the sequence value, in which case the memory location is cleared to zero. The sequence value simplifies the implementation of queues and circular buffers using atomic fetch-and-add operations, as pointers into arrays can be atomically updated and wrapped to the bounds of the arrays.

The hardware that performs atomic memory operations is integrated with the memory controllers that are distributed throughout the memory system. Effectively, each of the distributed atomic memory operation units is assigned a region of memory, and is responsible for performing all atomic memory operations on memory locations within its assigned region. Distributing the hardware that performs atomic memory operations allows communication and synchronization to be localized, and supports more concurrency and memory-level parallelism. This distributed organization provides several advantages. Most importantly, it ensures that for every memory location, there is a single serialization point for all atomic memory operations that are performed on that location. This is necessary for correctness, as there must be a linear serialization of atomic memory operations on a particular memory location for the operations to be of practical use. Distributing the atomic memory operation units allows them to be placed close to the memory locations they operate on, which improves performance by increasing locality and reducing communication bandwidth. Rather than fetching the value stored at a memory location to a processor and updating the value there, the update specified by an atomic memory operation is performed in the memory system close to the memory location referenced by the

operation. This eliminates at least one round trip between the memory location and the processor. Finally, the distributed organization scales well. The number of concurrent atomic memory operations the memory system can support scales with the number of concurrent memory transactions the system can support, which allows software to exploit extensive concurrency and parallelism within the memory system. Because memory is distributed with the processors, the distributed organization allows software to map threads and shared data objects so that synchronization message traffic remains local to the communicating processors.

Atomic memory operations may be performed on cached data provided that the data is not cached in multiple locations and the data is consistently accessed through the cache. When atomic memory operations operate on cached data, the operations are performed at a cache controller, and operate on a cached local copy of the data. The cache controller provides the synchronization point that orders concurrent atomic memory operations.

Initiating Computation

Elm uses a simple messaging mechanism to allow computation to be dispatched to a processor. The messages effectively encode a load-and-jump operation that is issued when the message is received. The two messages differ in priority. The high priority message interrupts the processor and effectively terminates any task that happens to be executing on the processor. The low priority message is only dispatched if the destination processor is idle when the message arrives. If the processor is not idle, the message is discarded and a negative acknowledgment is returned to the sender.

These messages are used as a mechanism for dispatching and distributing computation to processors. They allow Elm to implement a very limited form of active messaging as a mechanism for initiating computation at a processor [32, 146]. However, the lack of support for multiple execution contexts and priorities within a processor severely limits their general usefulness [34, 91]. Certain aspects of the Elm architecture require that software be explicitly constructed to allow the interrupted computation to be resumed. Consider the problem of resuming the interrupted thread when its instructions are displaced. The compiler and runtime system would need to enforce a convention for indicating what instruction register context should be restored when the dispatched instructions complete. Software could reserve some instruction registers for message handlers, though this convention would reduce the number of instruction registers that are available. A more efficient approach would be to place the address of an instruction block that will restore the instruction registers after the handler completes. The handler would transfer control to the instruction block, which would be responsible for restoring the instruction registers and transferring control to the instruction at which the thread was interrupted.

8.2 Microarchitecture

This section describes how the concepts discussed previously are reflected in specific aspects of the Elm microarchitecture.

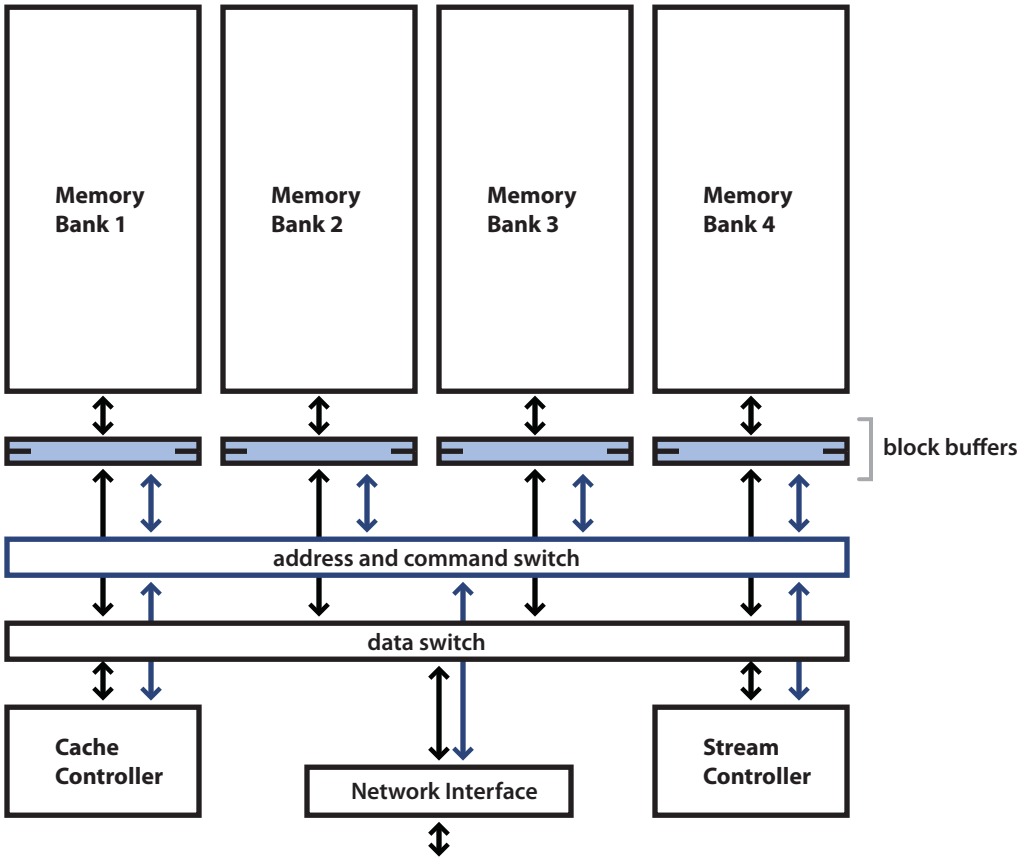


Figure 8.9 – **Memory Tile Architecture.** The memory is banked to allow multiple concurrent memory transactions to be serviced in parallel. The cache controller implements the cache protocols, and handles cached and atomic memory operations. The stream controller implements the stream transfer protocols and handles stream and block memory operations.

Memory Tile Architecture

Figure 8.9 illustrates the distributed memory tile architecture. The memory is banked to allow multiple concurrent memory transactions to be serviced in parallel. The banking allows dense and efficient single port memory arrays to be used. The cache controller implements the cache protocols, and handles cached and atomic memory operations. The stream controller implements the stream transfer protocols and handles stream and block memory operations.

The block buffers associated with the memory banks stage data transfers between the memory arrays and the switches. The block buffers decouple the scheduling of memory access operations from the allocation of switch bandwidth, which improves memory and switch bandwidth utilization. The block buffers also provide a simple mechanism for capturing spatial locality at the memory banks. They allow data read from a memory bank to be delivered to the switch over multiple cycles, and allow write data arriving from the switch to be

buffered before the data is delivered to the memory bank.

We often want the memory read and write operations to access more bits than the switch will accept or deliver in one cycle. We often prefer SRAM arrays with relatively short bit-lines because the bit-line capacitance that must be discharged when the memory is accessed is reduced, which improves access times. Reducing the number of cells connected to a bit-line also improves read reliability in modern technologies by reducing bit-line leakage. To reduce the number of rows while keeping capacity constant, we must increase the number of columns in the array. Rather than increasing the column multiplexing factor, it is often more efficient to reduce the number of times the bit-lines are cycled by capturing more of the data on the discharged bit-lines, temporarily buffering it in the block buffers, and transferring it to the switch when requested.

To improve further the area efficiency of the switch, the switch data-paths could be designed to operate at a faster clock rate than the control-paths; for example, the switch could transfer data on both the rising and falling edge of the clock.

Cache Controller

Figure 8.10 provides a more detailed illustration of the cache controller. The control logic block contains the control logic and finite state machines that implement the cache control protocol. It orchestrates the handling of cached memory operations, synthesizing the sequences of commands required to implement the operations. The configuration registers hold the cache configuration state, such as the addresses tables that specify the addresses of the memory controllers that are used to handle cache misses. The pending transaction registers hold the state used by the cache controller to track active cached memory transactions. The message scoreboard tracks pending memory operations, and notifies the control logic when messages that signal the completion of memory operations initiated by the cache module are handled. The atomic memory unit handles all atomic memory operations that arrive at the memory tile. The atomic memory unit is implemented in the cache controller to simplify the handling of cached atomic memory operations.

Rather than provide dedicated storage for cache tags and state information, the cache tags and associated block state bits are stored in the memory banks along with the block data. Distributed memory is expected to be used mostly as software-managed memory, and this organization avoids the expense of providing dedicated tag arrays. However, for applications that rely extensively on caching to improve locality, dedicated tag arrays are more efficient. The controller uses a small tag cache to capture recently accessed tags close to the control logic. The tag cache effectively exploits locality within cache blocks, and improves efficiency by filtering tag accesses that would otherwise access the memory banks. It also reduces cache access times when memory access streams exhibit spatial and short-term temporal locality by eliminating the bank access component of the tag access time.

The cache controller supports cache block and cache way locking. To avoid deadlock and simplify the design of the controller, one way of the cache cannot be locked. This ensures that the controller can always allocate a block when handling a cache miss. Cache block locking improves performance and efficiency when cached stream operations use indexed addressing modes. The indices are stored locally, as this reduces the index access latency seen by the stream controller. Consequently, the indices and stream data compete

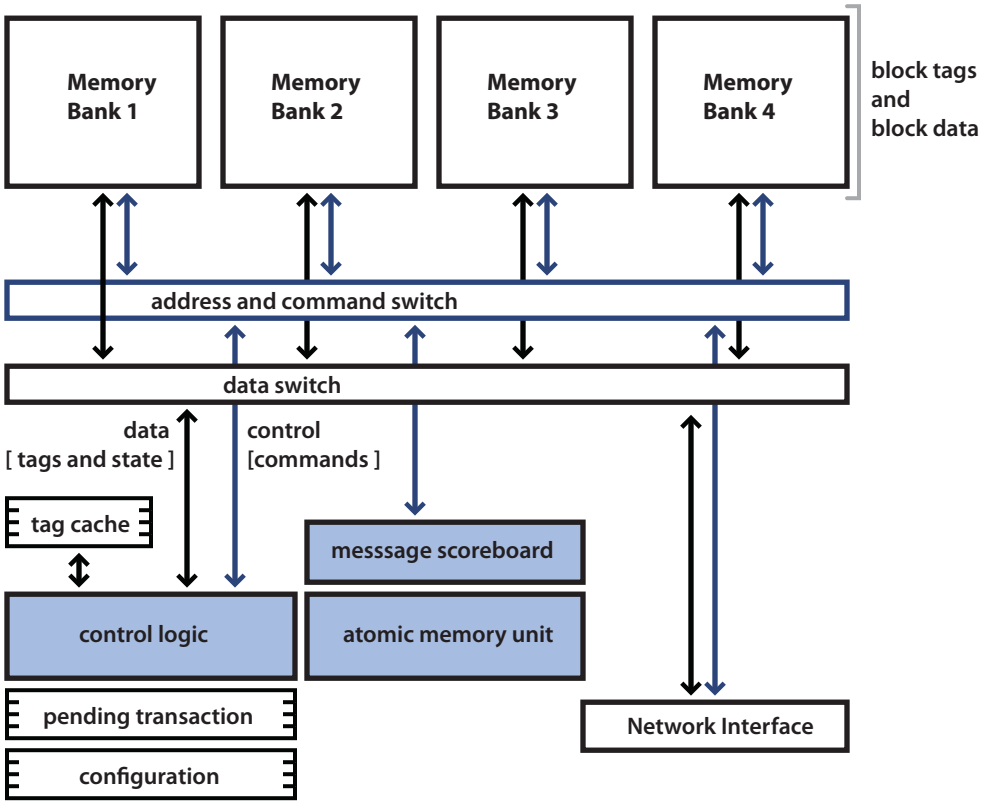


Figure 8.10 – **Cache Architecture.** The control logic implements the cache protocol and orchestrates the handling of cached memory operations. The cache tags and associated state are stored in the memory banks along with the data blocks. The tag cache implements a small cache for holding recently accessed tag and state data in the cache controller. The pending transaction registers hold the state used by the cache to track active cached memory transactions. The configuration registers hold the cache configuration state; this includes the mappings used to determine the addresses of the module that back the cache. The message scoreboard tracks pending memory operations, and notifies the control logic when messages that signal the completion of memory operations initiated by the cache module are handled. The atomic memory unit performs atomic memory operations.

for space in the cache. Locking the cache blocks that hold the indices prevents data blocks that are brought into the cache during the handling of stream memory operations from displacing the indices.

The cache controller allows software to explicitly invalidate individual blocks, ways, and caches. An invalidation operations forces the cache to release any cached copies of data blocks it holds. Blocks that have been modified are written to memory when they are released. This allows software to force data to be flushed to memory during software orchestrated transitions from private to shared states.

Stream Controller

Figure 8.11 provides a more detailed illustration of the stream controller. The record address generation unit computes the effective memory addresses of the records that are referenced in a stream operation: it provides the source addresses to stream load operations, and the destination addresses to stream store operations. The stream address generation unit computes the effective memory addresses of the packed stream elements: it provides the destination addresses to a stream load operation, and the source addresses to a stream store operation. The stream state update unit updates the stream state after a stream element transfer is dispatched.

The control logic block contains the control logic and finite state machines that implement the stream transfer protocol. It orchestrates the handling of stream memory operations, synthesizing the sequences of memory operations required to implement stream transfers. The message scoreboard tracks pending memory operations and notifies the control logic when messages that signal the completion of memory operations previously initiated by the stream controller are handled. The scoreboard also tracks which stream and transaction registers are in use, and the state of the associated stream memory operations.

The stream descriptor registers hold the stream descriptors, which are not modified by the stream controller. The stream state registers hold the active stream state, which is updated during a stream operation. A stream state register is updated when a stream memory operation transfers a record. The transaction registers hold state associated with pending stream memory operations. The stream controller associates a transaction register with each stream descriptor register, and uses that transaction register to handle stream memory operations that reference the stream descriptor register.

To improve the handling of stream memory operations that use indexed addressing modes, the stream controller uses an index buffer to cache elements of an index vector at the stream controller. The index vector access patterns exhibit significant spatial locality, and the index buffer allows blocks of elements to be prefetched and stored locally at the stream controller. The index buffers allow a single bank access to transfer multiple indices to the stream controller, which improves efficiency by amortizing the memory access that reads a block of indices across multiple record transfers. Rather than allocate an index buffer for each stream descriptor register, the stream controller dynamically allocates index buffers to active indexed stream transfers. This avoids the area expense of implementing an index buffer for each stream descriptor. The index buffers are invalidated after each stream memory operation, which ensures that the index buffers do not cache expired copies of index data across stream operations.

The stream controller also handles memory operations that transfer blocks of data. It handles block memory operations as a primitive stream memory operation that operates on an anonymous stream. Cached block memory operations are forwarded to the cache controller, using the same path that handles cached stream operations.

Implementation of Decoupled Loads and Stores

Decoupled load operations use the presence bits associated with registers in the general-purpose register file to indicate when there is a load operation pending. Decoupled load operations that reference addresses beyond the local memory space are translated into read request messages as they depart the Ensemble. The

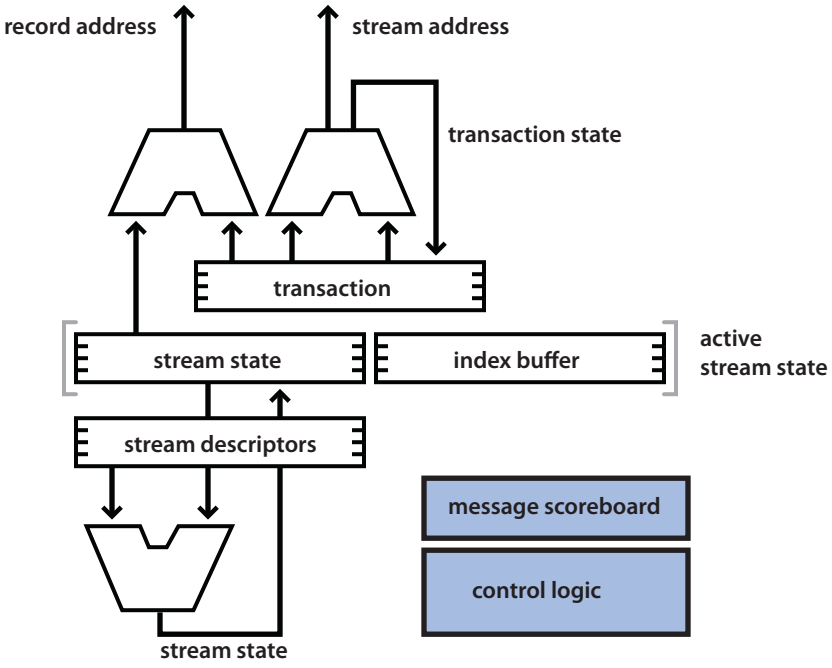


Figure 8.11 – **Stream Architecture.** The transaction registers hold state associated with pending stream memory operations. The stream controller associates a transaction register with each stream descriptor register, and uses that transaction register to handle stream memory operations that reference the stream descriptor register. The stream descriptor registers hold the stream descriptors, which are not modified by the stream controller. The stream state registers hold the active stream state, which is updated during a stream operation. The index buffer allows the stream controller to prefetch indices when handling indexed stream memory operations. The index vector access patterns exhibit significant spatial locality, and the index cache allows blocks of elements to be prefetched and stored locally at the stream controller. The message scoreboard tracks pending memory operations and notifies the control logic when messages that signal the completion of memory operations initiated by the stream controller are handled. The scoreboard also tracks which stream and transaction registers are in use, and the state of the associated stream memory operations.

register identifier is embedded in the source field of the message, which is used to route the returned load data. The memory controller that responds to the read request inserts the source address from the request address into the destination field of the response message. The local memory controller extracts the register identifier and forwards the load value and register identifier to the processor that issued the request. This convention uses the additional addresses that are available in the Ensemble address ranges to avoid having to allocate storage to record the return register addresses within the Ensemble memory controllers.

Hardware Memory Operations and Messages

This section provides a brief description how memory operations are handled. Elm allows most message classes to encode the number of data words contained in the payload section. This complicates the design

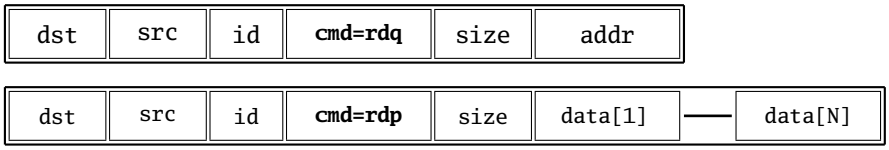


Figure 8.12 – Formats used by read Messages. The destination field (`dst`) encodes the address of the destination module to which the message is delivered; the source field (`src`) encodes the source address of the sending module. The identifier field (`id`) encodes an identifier that associates request and response message pairs. The size field (`size`) encodes the size of the block. The address field (`addr`) in the request message encodes the block address; the data fields (`data`) in the response message contains the block data.

of the interconnection network somewhat, but improves efficiency by avoiding the internal and external fragmentation problems associated with fixed message sizes.

Messages that are issued to handle load operations encode the load destination register in the source address field of the message. Essentially, some of the addresses within the address range assigned to a processor are used to encode register locations. This avoids the need to record the destination register associated with pending read requests. Instead, buffering in network is used to hold information about where data is to be returned; in effect, this allows the state storage for pending memory operations to scale with system size and memory access latencies.

Read Messages [read]

The request message requests [**cmd=rdq**] a block of contiguous words at a specified address from a memory controller; the response message [**cmd=rdp**] returns the data block read from the memory (Figure 8.12). The block size is encoded in the size field. Both messages provide an operation identifier field to allow the source controller to match the response message to the request. The message identifier is used to associate messages with pending state, such as the pending transaction state maintained in cache controllers.

Write Messages [write]

The request message [**cmd=wrq**] transfers a block of contiguous data to the destination memory controller; the response message [**cmd=wrp**] indicates the write operation has completed (Figure 8.13). The block size is encoded in the size field. Both messages have an operation identifier field to allow the source controller to match the response to the request. The message identifier is used to associate messages with pending state, such as the pending transaction state maintained in cache controllers.

Atomic Compare-and-Swap [atomic.cas]

The request message [**cmd=csq**] transfers the address, compare, and swap values to the destination memory controller; the response message [**cmd=csp**] transfers the value read at the address (Figure 8.14). Both messages have an operation identifier field to allow the source controller to match the response to the request.

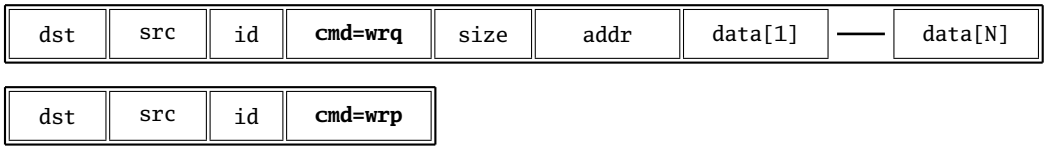


Figure 8.13 – **Formats used for write Messages.** The destination field (`dst`) encodes the address of the destination module to which the message is delivered; the source field (`src`) encodes the source address of the sending module. The identifier field (`id`) encodes an identifier that associates request and response message pairs. The size field (`size`) encodes the size of the block. The address field (`addr`) in the request message encodes the write address; the data fields (`data`) contain the block data.

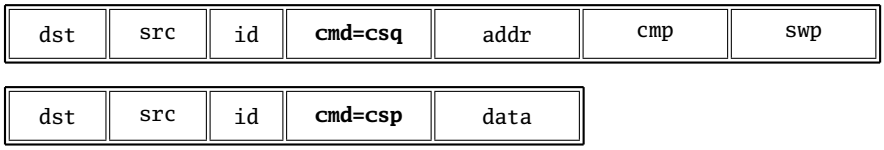


Figure 8.14 – **Formats used for compare-and-swap Messages.** The destination field (`dst`) encodes the address of the destination module to which the message is delivered; the source field (`src`) encodes the source address of the sending module. The identifier field (`id`) encodes an identifier that associates request and response message pairs. The address field (`addr`) encodes the address operand; the compare field (`cmp`) encodes the compare operand; the swap field (`swp`) encodes the swap operand; and the data field (`data`) encodes the value stored at the memory location when the request message was handled.

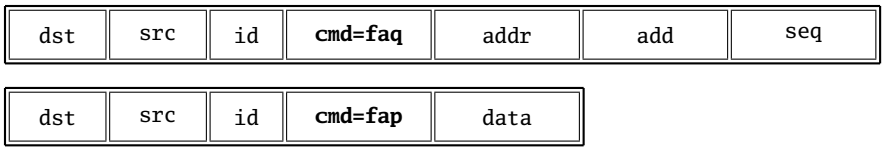


Figure 8.15 – **Formats used by fetch-and-add Messages.** The destination field (`dst`) encodes the address of the destination module to which the message is delivered; the source field (`src`) encodes the source address of the sending module. The identifier field (`id`) encodes an identifier that associates request and response message pairs. The address field (`addr`) encodes the address operand; the addend field (`add`) encodes the addend operand; and the sequence field (`seq`) encodes the sequence limit. The value stored at the specified memory address is set to zero if it matches the sequence limit. The data field (`data`) encodes the value stored at memory location when the request message was handled.

Atomic Fetch-and-Add [atomic.fad]

The request message [**cmd=faq**] transfers the address, update, and bound values to the memory controller; the response message [**cmd=fap**] transfers the value read at the address (Figure 8.15). Both messages have an operation identifier field to allow the source controller to match the response to the request.

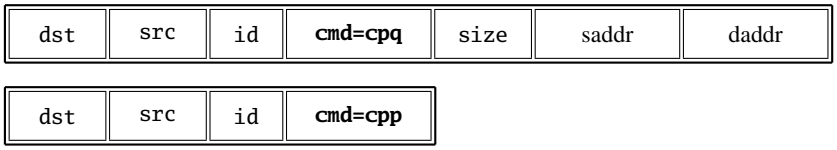


Figure 8.16 – **Formats of copy Messages.** The destination field (`dst`) encodes the address of the destination module to which the message is delivered; the source field (`src`) encodes the source address of the sending module. The identifier field (`id`) encodes an identifier that associates request and response message pairs. The size field (`size`) specifies the size of the block that will be copied. The source address field (`saddr`) encodes the source address to the copy operation; the destination address field (`daddr`) encodes the destination address to the copy operation.

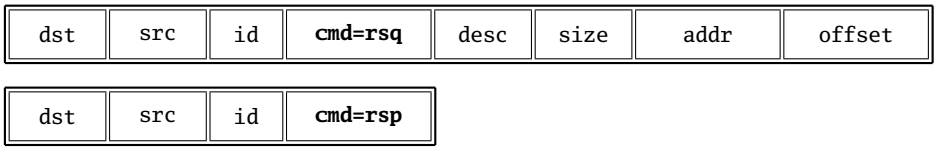


Figure 8.17 – **Formats of read-stream Messages.** The destination field (`dst`) encodes the address of the stream controller to which the message is delivered; the source field (`src`) encodes the source address of the sending module. The identifier field (`id`) encodes an identifier that associates request and response message pairs. The stream descriptor register is encoded in the descriptor field (`desc`). The size field (`size`) encodes the number of records to be transferred. The address field (`addr`) encodes the address to which records are transferred; the offset field (`offset`) encodes the offset that is used to compute the effective addresses of the records comprising the stream transfer.

Block Copy Messages [copy]

Block copy messages request that a block of data be transferred between two remote memory controllers. The request message [**cmd=cpq**] specifies the destination addresses of the copy operation and the number of words to be transferred; the response message [**cmd=cpq**] indicates that the transfer has completed (Figure 8.16). Both messages have an operation identifier field to allow the source controller to match the response to the request. Block copy operations are converted to read and write operations when they arrive at the destination memory controller. The memory controller performs the requested operation and returns a block copy response message when the operation completes.

Processors may use block copy messages to transfer data between remote memory locations without marshaling the data through a local memory. For example, a processor may send a block copy request to a distributed memory tile to instruct it to transfer data directly from system memory to distributed memory.

Read-Stream Messages [read-stream]

Read-stream messages initiate stream load operations. The stream load operation gathers records to a block of memory using a stream descriptor to generate the record addresses. The request message [**cmd=rsq**] specifies the stream descriptor register that will be used to generate the record addresses, the number of records to

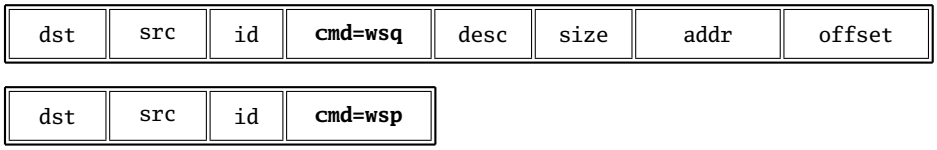


Figure 8.18 – **Formats of write-stream messages.** The destination field (`dst`) encodes the address of the stream controller to which the message is delivered; the source field (`src`) encodes the source address of the sending module. The identifier field (`id`) encodes an identifier that associates request and response message pairs. The stream descriptor register is encoded in the descriptor field (`desc`). The size field (`size`) encodes the number of records to be transferred. The address field (`addr`) encodes the address to which records are transferred; the offset field (`offset`) encodes the offset that is used to compute the effective addresses of the records comprising the stream transfer.

Figure 8.19 – **Formats of read-cache messages.** The destination field (`dst`) encodes the address of the cache controller to which the message is delivered; the source field (`src`) encodes the source address of the sending module. The identifier field (`id`) encodes an identifier that associates request and response message pairs. The size field (`size`) encodes the size of the block. The address field (`addr`) in the request message encodes the block address; the data fields (`data`) in the response message contains the block data.

be transferred, and the address at which the records are to be moved. The response message [**cmd=rsp**] is returned when the stream controller completes the operation. The stream controller orchestrating the operation issues sequences of read, write, and block copy messages to effect the stream transfer.

Write-Stream Messages [write-stream]

Write-stream messages initiate stream store operations. The stream store operation scatters records from a block of memory using a stream descriptor to generate the record addresses. The request message [**cmd=wsq**] specifies the stream descriptor register that will be used to generate the record addresses, the number of records to be transferred, and the address from which the records are to be moved. The response message [**cmd=wsp**] is returned when the stream controller completes the operation. The stream controller orchestrating the operation issues sequences of read, write, and block copy messages to effect the stream transfer.

Read-Cache Messages [read-cache]

These messages load blocks of data using a distributed memory as block cache. The request message [**cmd=rdcq**] transfers the block address and block size to a cache controller; the response message [**cmd=rdcp**] returns the data block (Figure 8.19).

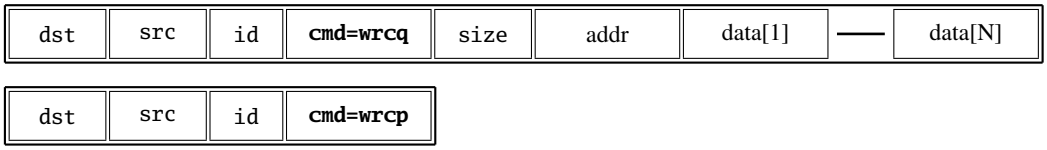


Figure 8.20 – **Formats of write-cache messages.** The destination field (`dst`) encodes the address of the cache controller to which the message is delivered; the source field (`src`) encodes the source address of the sending module. The identifier field (`id`) encodes an identifier that associates request and response message pairs. The size field (`size`) encodes the size of the block. The address field (`addr`) in the request message encodes the write address; the data fields (`data`) contain the block data.

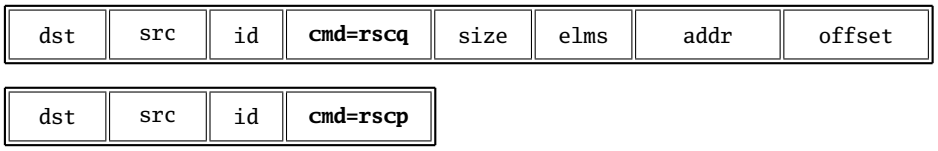


Figure 8.21 – **Formats of read-stream-cache messages.** The destination field (`dst`) encodes the address of the stream controller to which the message is delivered; the source field (`src`) encodes the source address of the sending module. The identifier field (`id`) encodes an identifier that associates request and response message pairs. The stream descriptor register is encoded in the descriptor field (`desc`). The size field (`size`) encodes the number of records to be transferred. The address field (`addr`) encodes the address to which records are transferred; the offset field (`offset`) encodes the offset that is used to compute the effective addresses of the records comprising the stream transfer.

Write-Cache Message [write-cache]

The messages store blocks of data using a distributed memory as a block cache. The request message [**cmd=wrcq**] transfers the data block to the distributed memory and specifies the write address; the response message [**cmd=wrcp**] indicates that the write operation has completed (Figure 8.20).

Stream-Read-Cache Messages [read-stream-cache]

These messages initiate stream load operations that use distributed memory tiles to cache stream records. The stream load operation gathers records to a block of memory using a stream descriptor to generate the record addresses. The request message [**cmd=rscq**] specifies the stream descriptor register that will be used to generate the record addresses, the number of records to be transferred, and the address at which the records are to be moved. The response message [**cmd=rscp**] is returned when the stream controller completes the operation. The stream controller orchestrating the operation issues sequences of read, write, and block copy messages that are filtered through its local cache controller to effect the stream transfer (Figure 8.21).

Stream-Write-Cache Messages [write-stream-cache]

These messages initiate stream store operations that use distributed memory tiles to cache records. The stream store operation scatters records from a block of memory using a stream descriptor to generate the

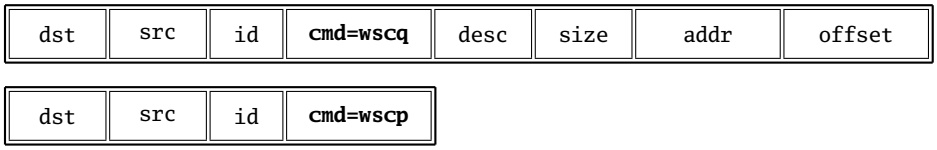


Figure 8.22 – **Formats of write-stream-cache messages.** The destination field (`dst`) encodes the address of the stream controller to which the message is delivered; the source field (`src`) encodes the source address of the sending module. The identifier field (`id`) encodes an identifier that associates request and response message pairs. The stream descriptor register is encoded in the descriptor field (`desc`). The size field (`size`) encodes the number of records to be transferred. The address field (`addr`) encodes the address to which records are transferred; the offset field (`offset`) encodes the offset that is used to compute the effective addresses of the records comprising the stream transfer.

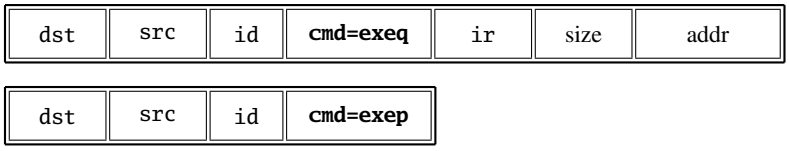


Figure 8.23 – **Format of execute Command Messages.** The destination field (`dst`) of the request message encodes the address of the processor to which the instructions are dispatched; the source field (`src`) encodes the address of the processor that issued the command. The identifier field (`id`) is used to associate request and response messages. The destination instruction register field (`ir`) encodes the instruction register at which the instructions are loaded, which determines the address control transfers to when the message is handled. The size field (`size`) encodes the number of instructions to be loaded, and the address field (`addr`) encodes the address of the instruction block in memory.

record addresses. The request message [**cmd=wscq**] specifies the stream descriptor register that will be used to generate the record addresses, the number of records to be transferred, and the address from which the records are to be moved. The response message [**cmd=wscp**] is returned when the stream controller completes the operation. The stream controller orchestrating the operation issues sequences of read, write, and block copy messages that are filtered through its local cache controller to effect the stream transfer (Figure 8.22).

Execute and Dispatch Messages [execute and dispatch]

Execute operations allow the sender to dispatch an execute command at the receiver. The execute command tells the receiving processor to begin executing the instructions at the specified address. The request message [**cmd=exeq**] specifies the address (`addr`) and size (`size`) of an instruction block and the instruction register address the block is copied to (`ir`) within the receiving processors. The processor transfers control to the instruction register specified in the message. The execute response message [**cmd=exep**] is returned to indicate that the processor has loaded the instructions.

Dispatch messages transfer the instruction block in the message (Figure 8.24). This allows the instructions to be dispatched without creating a local copy of the instruction blocks.

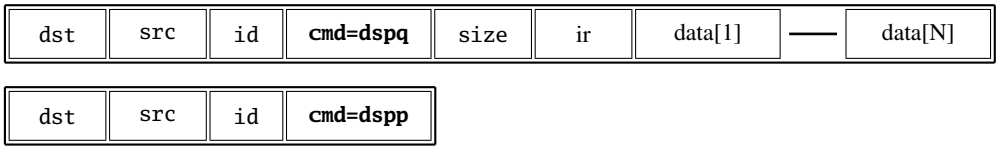


Figure 8.24 – **Format of dispatch Command Messages.** The destination field (`dst`) of the request message encodes the address of the processor to which the instructions are dispatched; the source field (`src`) encodes the address of the processor that issued the command. The identifier field (`id`) is used to associate request and response messages. The size field (`size`) encodes the number of instructions in the message. The destination instruction register field (`ir`) encodes the instruction register at which the instructions are loaded, which determines the address control transfers to when the message is handled. Instructions are transported in the data fields.

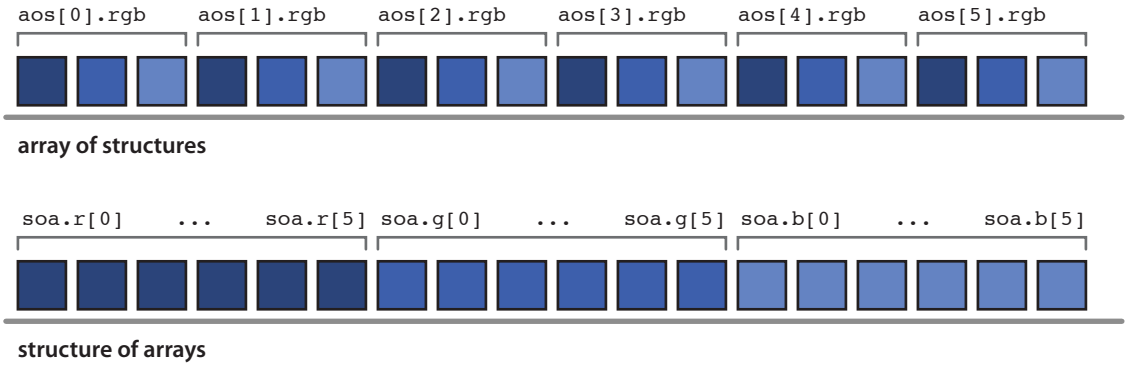


Figure 8.25 – **Examples of Array-of-Structures and Structure-of-Arrays Data Types.**

8.3 Example

The following example illustrates the use of several mechanisms provided by the Elm memory system.

Array-of-Structures and Structure-of-Array Conversion

Applications that process sequences of aggregate data objects usually represent collections of aggregates as arrays-of-structures and structures-of-arrays. Figure 8.25 illustrates the difference in data layout that arises from representing a collection of an aggregate data type as an array-of-structures and a structure-of-arrays. The array-of-structures representation exposes spatial locality within the individual aggregate data types. The structure-of-arrays representation exposes spatial locality within the fields of the aggregate data type. Machines that provide instructions that operate on multiple data elements in parallel usually prefer data that is organized as structures-of-arrays, as the structure-of-arrays representation simplifies the mapping of data elements to parallel function units. Stream load and store operations allow sequences of aggregates to be converted between array-of-structures and structure-of-arrays representation efficiently.

Figure 8.26 shows a code fragment that uses scalar load and store operations to convert between array-of-structure and structure-of-array representations. The code fragment is relatively expensive. It uses registers to

```
100 struct { fixed32 r;    fixed32 g;    fixed32 b;   } aos[6];
101 struct { fixed32 r[6]; fixed32 g[6]; fixed32 b[6] } soa;
102
103 // array-of-structures to structure-of-arrays
104 for ( int j = 0; j < 6; j++ ) {
105   soa.r[j] = aos[j].r;
106   soa.g[j] = aos[j].g;
107   soa.b[j] = aos[j].b;
108 }
109
110 // structure-of-arrays to array-of-structures
111 for ( int j = 0; j < 6; j++ ) {
112   aos[j].r = soa.r[j];
113   aos[j].g = soa.g[j];
114   aos[j].b = soa.b[j];
115 }
```

Figure 8.26 – Scalar Conversion Between Array-of-Structures and Structure-of-Arrays.

```
100 struct { fixed32 r;    fixed32 g;    fixed32 b;   } aos[6];
101 struct { fixed32 r[6]; fixed32 g[6]; fixed32 b[6] } soa;
102 stream_descriptor sd = { 1,   // stream record-size
103                          6,   // number of records in stream
104                          1,   // stream element stride
105                          0 }; // index address (null)
106
107 // array-of-structures to structure-of-arrays
108 stream_load( &soa.r, &aos.r, &sd );
109 stream_load( &soa.g, &aos.g, &sd );
110 stream_load( &soa.b, &aos.b, &sd );
111 stream_sync();
112
113 // structure-of-arrays to array-of-structures
114 stream_store( &aos.r, &soa.r, &sd );
115 stream_store( &aos.g, &soa.g, &sd );
116 stream_store( &aos.b, &soa.b, &sd );
117 stream_sync();
```

Figure 8.27 – Stream Conversion Between Array-of-Structures and Structures-of-Arrays.

stage data and function units to orchestrate the movement of individual elements. The number of instructions that are executed increases in proportion to the number of data elements that are moved, which is the product of the array length and the number of elements in the aggregate data type.

Figure 8.27 shows an equivalent code fragment that uses stream load and store operations to convert between array-of-structure and structure-of-array representations. Data is moved between memory locations without passing through registers. The function units are used to initiate and coordinate the synchronization of the data movement, but are not used to orchestrate the movement of individual elements. The number of instructions that are executed increases in proportion to the number of elements in the aggregate data type, and is independent of the array length.

	Execution Time [cycles]	Messages Exchanged [messages]	Data Transferred [words]
Remote Loads and Stores [Kernel 1]	1,728	256	896
Decoupled Loads and Stores [Kernel 2]	472	256	896
Block Gets and Puts [Kernel 3]	736	32	224
Decoupled Block Gets and Puts [Kernel 4]	(480) 498	32	224
Stream Loads and Stores [Kernel 5]	451	(4) 32	(140) 224

Table 8.1 – Execution Times and Data Transferred. The rows list the execution times, messages exchanged, and data transferred for each of the kernel implementations. The tabulated data are reported per kernel iteration. The data transferred includes all of the data comprising the messages exchanged between the local and remote memory controllers, and includes addresses and command information exchanged by the memory controllers. Parenthetical entries list the lower limits that are achieved with more aggressive software and hardware optimizations that allow multiple records to be transferred in a single memory system message.

8.4 Evaluation and Analysis

This section evaluates the impact of mechanisms provided by the Elm memory system on the efficiency of several embedded benchmark codes. The benchmarks are compiled and mapped to 16 processors within 4 Ensembles. Data are transferred between kernels using Ensemble memory at the destination processor to receive and buffer data.

Remote Memory Access Efficiency

To evaluate the impact that the different remote memory access operations have on the performance and memory bandwidth demands of kernels that operate on data residing in remote memory, we can analyze several implementations of an image processing kernel that applies a filter to multiple square regions of an image. We assume that the kernel is passed a sequence of independent regions that are to be filtered. The image resides in remote memory and is updated in place, so the kernel must fetch elements of the image, operate on them, and then store the updated elements back to the image. Figure 8.28 provides a simplified illustration of the sequence of operations. To simplify the example, we assume that the filter operation depends only on a single pixel and that the three color components comprising a pixel are packed in a single memory word. As the details of the filter operation are unimportant, we represent the operation as a generic `filter` function.

To make the example concrete, we shall assume that the filter operation consumes 5 cycles, that transferring a message between the processor and remote memory consumes 5 cycles, and that all local memory operations complete in 1 cycle. This results in remote load and store operations requiring 11 cycles to complete: 5 cycles to send the request message, 1 cycle to perform the memory operation at the remote memory, and 5 cycles to send the response message. Table 8.1 summarizes the execution times, the numbers of messages exchanged, and the data transferred for the different implementations of the kernels.

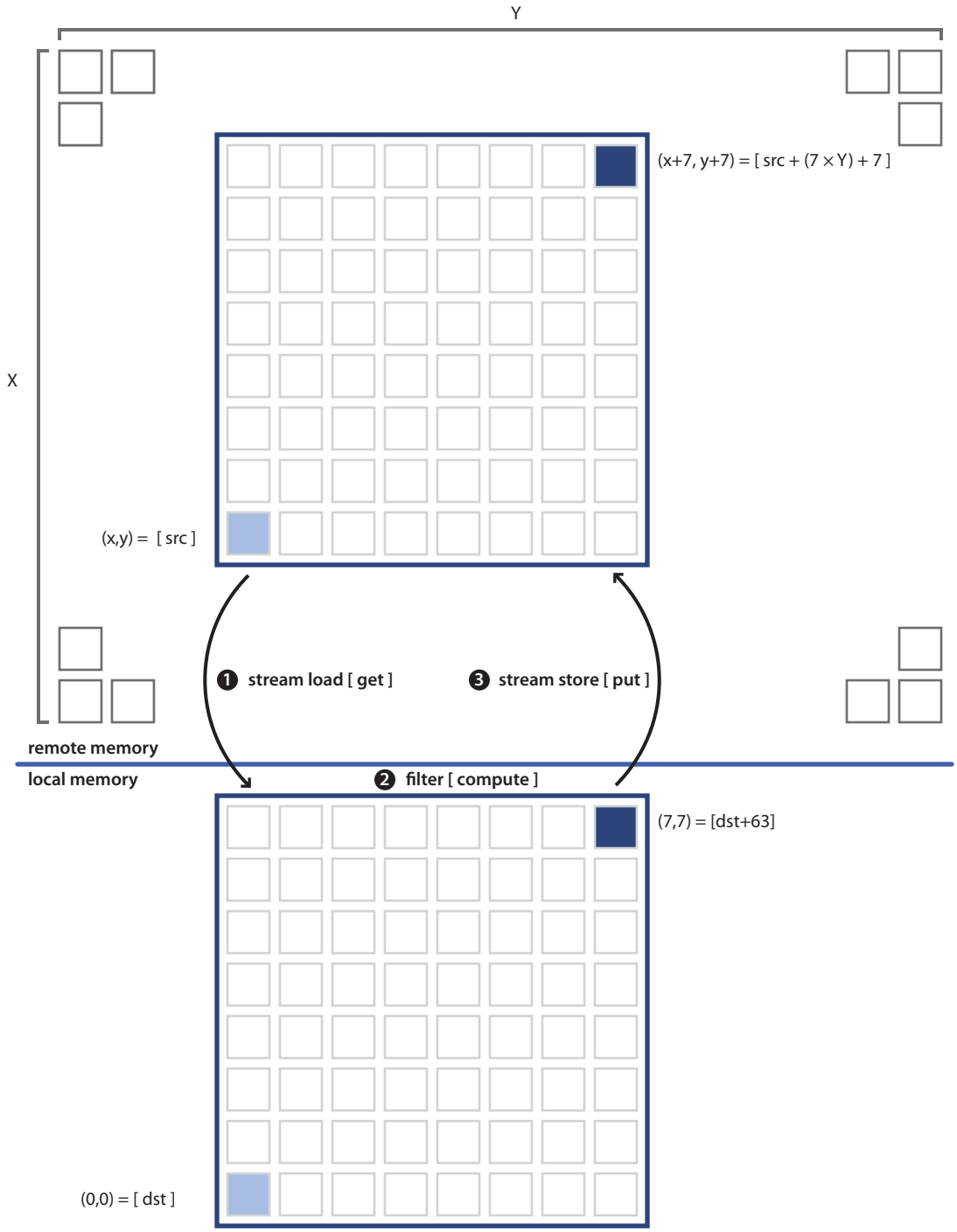


Figure 8.28 – **Image Filter Kernel.** The elements comprising the region of the image to be updated is transferred to local memory, where the region is filtered in place. After the filter operation completes, the updated region is transferred back to the image.

```
100   for ( int n = 0; n < size; n++ ) {
101     int x = pos[n].x;
102     int y = pos[n].y;
103     for ( int i = 0; i < 8; i++ ) {
104       for ( int j = 0; j < 8; j++ ) {
105         image[y+i][x+j] = filter( image[y+i][x+j] ) ;
106       }
107     }
108   }
```

Figure 8.29 – Implementation using Remote Loads and Stores [Kernel 1]. The kernel accepts a sequence of regions that are to be processed through the `pos` and `size` variables. Iteration n of the kernel applies the filter to the 8×8 block of pixels located at the pixel coordinate specified by `pos[n]`.

Remote Loads and Stores [Kernel 1]

The simplest implementation of the kernel uses remote loads and stores to transfer elements of the image between remote memory and registers. Figure 8.29 shows a code fragment that implements the kernel using remote load and store operations. The image is updated in place, and individual elements of the image are accessed using remote load and store operations. Each load operation generates a remote read message, and each store operation generates a remote write message. The execution time and data transferred are listed in the first row of Table 8.1.

Decoupled Remote Loads and Stores [Kernel 2]

We can improve the performance of the kernel by using decoupled memory operations to allow multiple memory operations to proceed in parallel, and to allow memory operations to complete in parallel with computation. Figure 8.29 shows a code fragment that implements the image filter using decoupled remote load and store operations. Elements of the original and filtered image are temporarily stored in registers to decouple the handling of memory operations from the computation.

The execution time and data transferred are listed in the second row of Table 8.1. The decoupling improves performance, but does not affect the number of memory transactions and the amount of data transferred. The decoupling consumes additional registers. Because the reduction in the execution time is less than the increase in register utilization, the decoupling increases register utilization and aggregate register pressure.

Block Gets and Puts [Kernel 3]

We can reduce the number of messages that are exchanged by using block get and put operations to transfer blocks of elements between remote memory and local memory. By exploiting spatial locality within blocks of elements to reduce the number of remote transfers, we amortize the overhead of transferring messages between the local and remote memory controllers across multiple elements. Consequently, the demand for bandwidth in the interconnection network is reduced, and less energy is expended moving data. Figure 8.31 shows a code fragment that implements the image filter using block get and put operations. Each iteration

```
100  for ( int n = 0; n < size; n++ ) {
101    int x = pos[n].x;
102    int y = pos[n].y;
103    fixed32 ti[8], to[8];
104    for ( int i = 0; i < 8; i++ ) {
105      for ( int j = 0; j < 8; j++ ) {
106        ti[j] = image[y+i][x+j];
107      }
108      for ( int j = 0; j < 8; j++ ) {
109        to[j] = filter( ti[j] );
110      }
111      for ( int j = 0; j < 8; j++ ) {
112        image[y+i][x+j] = to[j];
113      }
114    }
115  }
```

Figure 8.30 – **Implementation using Decoupled Remote Loads and Stores [Kernel 2].** Temporary variables `ti` and `to` provide local storage for staging transfers from and to remote memory. The temporary variables are mapped to registers, and the remote memory accesses are implemented as decoupled remote load and store operations.

```
100  for ( int n = 0; n < size; n++ ) {
101    int x = pos[n].x;
102    int y = pos[n].y;
103    fixed32 ti[8], to[8];
104    for ( int i = 0; i < 8; i++ ) {
105      block_get( &ti, &image[y+i][x], 8);
106      for ( int j = 0; j < 8; j++ ) {
107        to[j] = filter( ti[j] );
108      }
109      block_put( &to, &image[y+i][x], 8);
110    }
111  }
```

Figure 8.31 – **Implementation using Block Gets and Puts [Kernel 3].** Temporary variables `ti` and `to` provide local storage for staging transfers from and to remote memory. Local memory is used to stage remote memory accesses, and the temporary variables are mapped to local memory. Data are transferred between local and remote memory using get and put operations; data are transferred between local memory and registers using load and store operations.

transfers one row of the update region to local memory, performs the filter operation locally, and then transfers the filtered row back to the image.

The execution time and data transferred are listed in the third row of Table 8.1. The block gets and puts exploit spatial locality within the rows of the update region. Using block gets and puts to transfer data between remote memory and local memory reduces the number of remote memory transactions and messages. The reduction in the number of memory transactions achieves an execution time that is less than the version of the code that uses remote loads and stores. However, the failure to overlap memory accesses and computation results in an execution time that is greater than the version that uses decoupled loads and stores.

```
100  for ( int n = 0; n < size; n++ ) {
101    int x = pos[n].x;
102    int y = pos[n].y;
103    fixed32 block[8][8];
104    block_get_decoupled( &block[0][0], &image[y][x], 8);
105    for ( int i = 0; i < 8; i++ ) {
106      block_get_sync();
107      if ( i < 7 ) {
108        block_get_decoupled( &block[i+1][0], &image[y+i+1][x], 8);
109      }
110      for ( int j = 0; j < 8; j++ ) {
111        block[i][j] = filter( block[i][j] );
112      }
113      block_put_decoupled( &block[i][0], &image[y+i][x], 8);
114    }
115    block_put_sync();
116  }
```

Figure 8.32 – Implementation using Decoupled Block Gets and Puts [Kernel 4]. Temporary variable `block` provides a name for staging transfers from and to remote memory. Local memory is used to stage remote memory accesses, and the elements of temporary variable `block` are mapped to local memory. Data are transferred between local and remote memory using get and put operations; data are transferred between local memory and registers using load and store operations. Blocks of data are fetched one iteration before being accessed. Synchronization operations are inserted to ensure that data transfers have completed before local copies of data are accessed, and to ensure that all outstanding store operations have completed before the kernel completes.

Decoupled Block Gets and Puts [Kernel 4]

As before, we can improve the performance of the kernel by using decoupled block get and put operations to allow memory operations to proceed in parallel with computation. We must explicitly synchronize the computation with the block memory transfers, which produces a small increase in the number of instructions that are executed. Figure 8.32 shows a code fragment that implements the image filter using decoupled block get and put operations. The code fetches the next row of the update region before performing the filter operation, which allows the memory access latency to be hidden by computation.

The execution time and data transferred are listed in the fourth row of Table 8.1. The block gets and puts exploit spatial locality within the rows of the update region, and decoupling allows memory operations to complete in parallel with computation. The overhead of issuing explicit memory and synchronization operations results in an execution time that is slightly greater than the version that uses decoupled loads and stores. However, this version of the code generates considerably fewer messages and reduces the data exchanged between the local memory and remote memory by a factor of four.

The execution time shown in parenthesis corresponds to an implementation that issues multiple concurrent block get operations to fetch the next update region before processing the current update region. This increases the fraction of the memory access time that can be overlapped with computation. The time required to transfer the first update region is exposed, but subsequent remote memory access latencies are completely hidden by the filter computation. This also reduces the number of synchronization operations that need to be

```
100  stream_descriptor sds = sdl = { 8,   // stream record-size
101                                  8,   // number of records in stream
102                                  X,   // stream element stride
103                                  0 }; // index address (null)
104  int k = 0;
105  fixed32 block[3][8][8];
106  stream_descriptor_load( &image, sdl );
107  stream_descriptor_load( &image, sds );
108  stream_load( &block[k], &image[ pos[0].x ][ pos[0].y ], sdl);
109  for ( int n = 0; n < size; n++ ) {
110    int x = pos[n].x;
111    int y = pos[n].y;
112    stream_load_sync(sdl);
113    if ( n < size - 1 ) {
114      stream_load( &block[(k+1)%3], &image[ pos[n+1].x ][ pos[n+1].y ], sdl);
115    }
116    for ( int i = 0; i < 8; i++ ) {
117      for ( int j = 0; j < 8; j++ ) {
118        block[k][i][j] = filter( block[k][i][j] );
119      }
120    }
121    stream_store( &block[k], &image[x][y], sds);
122    k = (k+1)%3;
123  }
124  stream_store_sync(sds);
```

Figure 8.33 – Implementation using Stream Loads and Stores [Kernel 5]. The stream load and store operations use stream descriptors `sdl` and `sds` to generate the sequences of addresses that correspond to an 8×8 region of the image. The stream descriptor specifies a record size of 8 elements, which corresponds to one row of the 8×8 region, and a stride of X, which corresponds to the number of elements in one row of the image. The stream descriptor that is used to load the region to local memory is loaded at a stream descriptor register at the memory controller that is nearest to the image, as this allows the memory operations comprising the load operations to be generated locally. Similarly, the stream descriptor that is used to store the region to remote memory is loaded at a stream descriptor register at the local memory controller, as this allows the memory operations comprising the stream store operation to be generated locally. Temporary variable `block` provides a name for staging transfers between local and remote memory. The kernel processes three regions of the image concurrently, with the computation of one region proceeding in parallel with the transfer of the previous region from local to remote memory and the transfer of the next region from remote to local memory.

issued to one per region. Essentially, we use additional local storage to coarsen the basic unit of communication and thereby reduce both communication and synchronization overheads.

Stream Loads and Stores [Kernel 5]

Figure 8.33 shows a code fragment that implements the image filter using stream load and store operations. The code transfers an update region to local memory, performs the filter operation on the local copy, and then transfers the updated region back to the image. The code uses three local region buffers to decouple stream loads, computation, and stream stores.

The execution time and data transferred are listed in the fifth row of Table 8.1. This version achieves the lowest execution time and requires the fewest messages exchanged and data transferred. However, the extensive decoupling results in this version requiring the most local memory to stage transfers between local memory and remote memory.

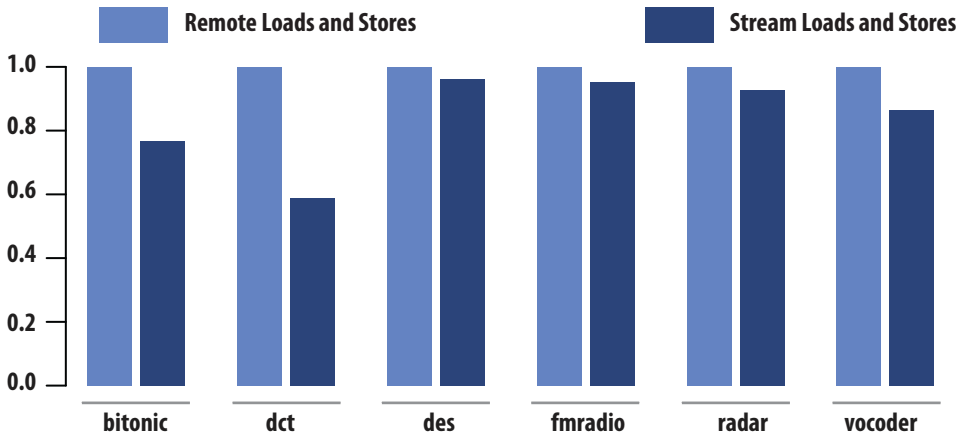


Figure 8.34 – **Normalized Kernel Execution Times.** The execution times are normalized within each kernel to illustrate the impact of the stream memory operations on the execution times.

The values listed in parenthesis under the messages exchanged and data transferred columns correspond to minimum values that would be achieved if all of the elements transferred by a stream memory operation were sent as a single read or write message. Most system would break the read and write operations into sequences of reads and write messages that each transfer a block of elements comprising one record of the stream.

Benchmark Throughput

Figure 8.34 illustrates the impact of the stream memory operations on throughput and latency. The figure compares the normalized execution times observed when the benchmarks are implemented using remote loads and stores to transfer data between kernels and when the benchmarks are implemented using stream loads and stores. Implementing remote memory transfers using stream memory operations reduces the execution time by 5% – 41%.

As we should expect, the improvement in execution time depends on the rate at which remote memory accesses are performed. Figure 8.35 shows the aggregate memory bandwidth demand for the benchmarks. The aggregate demand combines the memory operations performed by all 16 processors, which is why the demand sometimes exceeds one word per cycle. The component of the bandwidth that corresponds to local memory references is shown as separate from the component that corresponds to remote memory references. As Figure 8.34 and Figure 8.35 show, benchmarks that perform relatively few remote memory accesses exhibit small improvements in performance when stream memory operations are used to transfer data.

With the exception of the **fmradio** benchmark, the increase in memory bandwidth demand we observe in Figure 8.35 for the implementations that use stream memory operations is mostly caused by the reduction in the benchmark execution times. Figure 8.36 shows the number of memory accesses by component for each of the benchmarks. The reported data are normalized within each benchmark to illustrate the impact of stream memory operations. As the figure illustrates, the **fmradio** benchmark exhibits a small increase in the

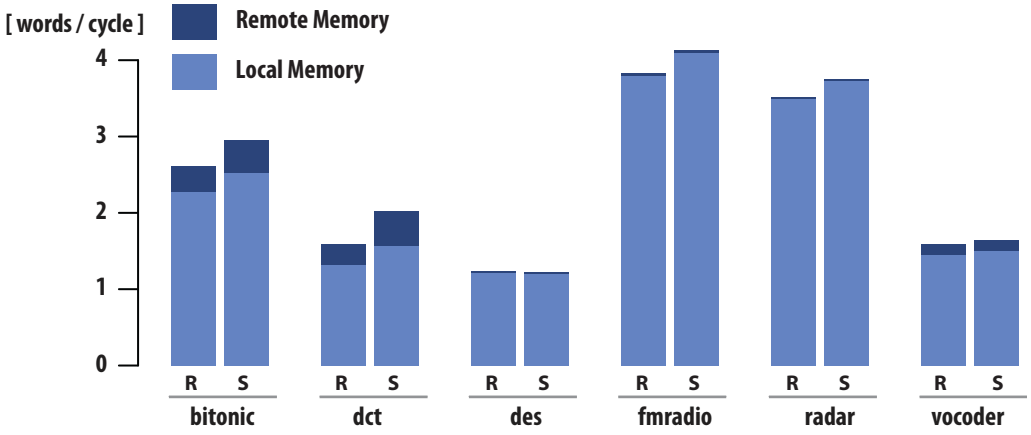


Figure 8.35 – **Memory Bandwidth Demand.** Remote (**R**) Loads and Stores. Stream (**S**) Loads and Stores. The data shows the application bandwidth demand when the applications are partitioned and mapped to 16 processors.

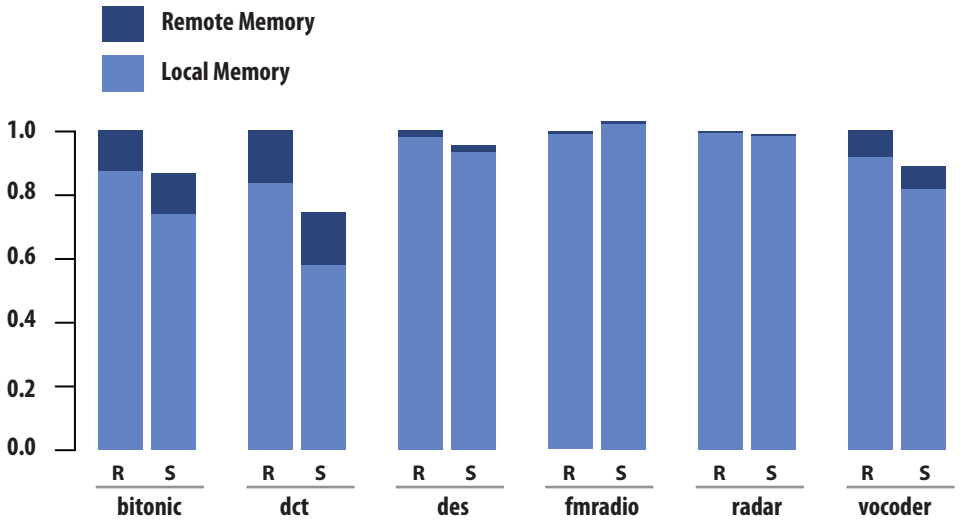


Figure 8.36 – **Normalized Memory Access Breakdown.** Remote (**R**) Loads and Stores. Stream (**S**) Loads and Stores. The memory accesses are normalized within each application to illustrate the breakdown of local and remote memory accesses.

number of local memory accesses. These additional accesses are contributed by store operations that place data in local memory where it is temporarily buffered before a stream store operation is issued to send the data to the receiving processor.

Figure 8.37 shows the reduction in the number of messages that traverse the interconnection network when communication between threads is implemented using stream memory operations. The implementations of the **radar** and **vocoder** benchmarks that use stream memory operations include a significant number

Remote Memory Operations					
bitonic	**dct**	**des**	**fmradio**	**radar**	**vocoder**
8,852	7,361	5,542	18,136	50,261	788,773

Stream Memory Operations					
bitonic	**dct**	**des**	**fmradio**	**radar**	**vocoder**
(16, 1) 6,673	(16, 1) 2,766	(8, 1) 260	(8, 1) 4,320	(1, 2) 25,123	(1, 1) 316,994
(32, 1) 1,550	(32, 1) 1,085	(16, 1) 715	(16, 1) 3,254	(2, 2) 1,049	(2, 1) 92,624
		(64, 1) 110	(64, 1) 120	(4, 2) 526	(8, 1) 3,598
		(16, 2) 282		(32, 2) 198	(15, 1) 7,710
		(64, 2) 37		(64, 2) 105	(20, 1) 4,112
					(96, 1) 2,569
					(1, 2) 63,909

Table 8.2 – Number of Remote and Stream Memory Operations in Benchmarks. The stream operation entries use the syntax (e,w) to indicate that the stream operation transfers a sequence of e elements, with each element comprising w contiguous words.

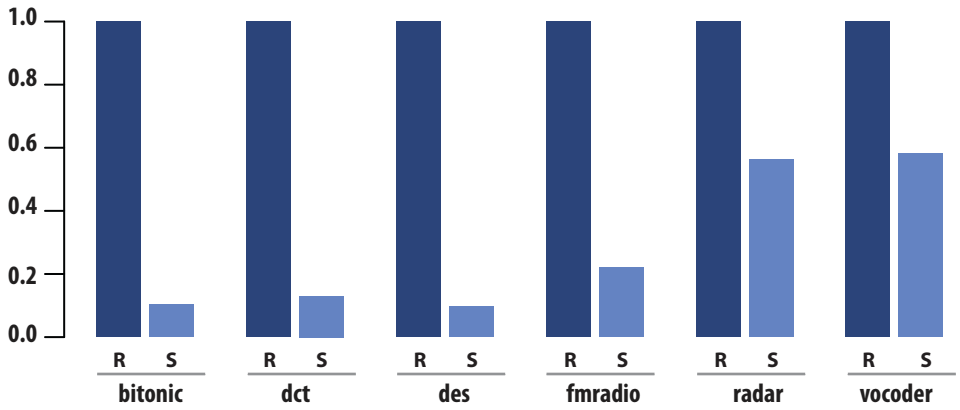


Figure 8.37 – Normalized Memory Messages. (**R**) Remote Loads and Stores. (**S**) Stream Loads and Stores. The data presented in the figure are normalized to illustrate the reduction in the number of messages that are communicated when the applications are implemented using stream load and store operations.

of remote load and store operations that access scalar variables. Table 8.2 lists the number of remote memory operations and stream memory operations performed in the implementations that use stream memory operations. Most of the scalar operations reference coordination variables that are used to synchronize the execution of the kernels and to interleave the enqueuing and dequeuing of data at distributed queues; other operations reference scalar data that is distributed across multiple Ensembles. The scalar references account for the smaller reduction in the number of messages we observe for these benchmarks in Figure 8.37.

Remote load and store operations produce messages that transport a single word of application data. Most stream memory operations produce messages that transport multiple words of application data, which amortizes the overhead of the message header over more application data. Figure 8.37 shows the the amount of data that traverses the interconnection network, which includes both application data, message header data, and acknowledgment messages sent in response when required by the message communication protocol. The

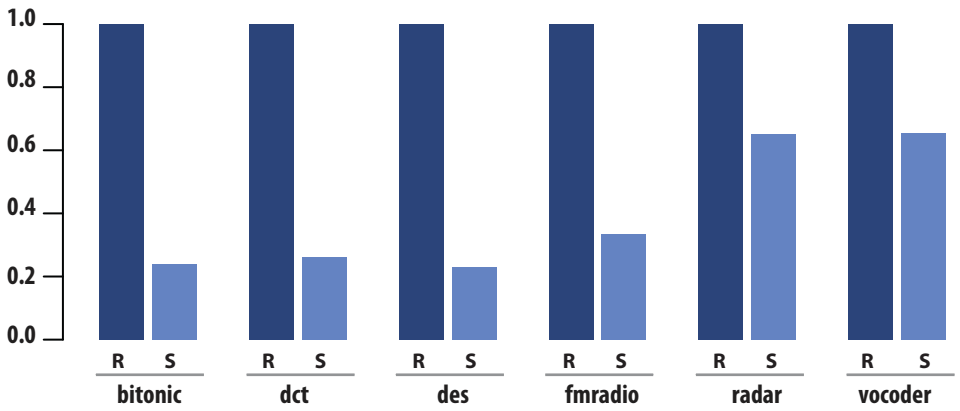


Figure 8.38 – Normalized Memory Message Data. Remote (**R**) Loads and Stores. Stream (**S**) Loads and Stores. The data presented in the figure are normalized to illustrate the reduction in the aggregate message data that is transferred when the applications are implemented using stream load and store operations.

reported data is normalized within each benchmark to illustrate the reduction achieved using stream memory operations. Comparing Figure 8.37 and Figure 8.38, we should observe that the reduction in message data is always less than the reduction in the number of messages. This is because the use of stream memory operations has little effect on the amount of application data that moves through the memory system. Rather, stream memory operations reduce the communication overheads associated with data movement.

Decoupled Memory Accesses

The availability of registers for receiving decoupled loads limits the extent to which decoupled memory operations can be used to tolerate remote memory access times. A straightforward application of Little's law [100] allows one to determine the number of registers that are needed to cover an expected memory access time and bandwidth demand. For example, consider a code fragment that demands one load operation every 5 cycles and has an effective load bandwidth of 0.2 words per cycle. If the expected memory access time is 20 cycles, the number of registers required to decouple memory accesses is $20 \times 0.2 = 4$ registers. If the memory access time is instead 100 cycles, the number of registers required to decouple memory accesses increases to $100 \times 0.2 = 20$ registers.

The Elm memory system uses the distributed and local Ensemble memories to stage data and instruction transfers. The Ensemble memories provide enough capacity to stage transfers to and from remote memories while reserving space for capturing local instruction and data working sets. The distributed memories provide additional capacity for staging transfers between local Ensemble memories and off-chip memory. They are also used to stage transfers between processors and dedicated IO interfaces. The decoupled stream memory operations provide an effective way of hiding communication latencies, and they reduce the number of pending transactions that the processors need to monitor and track. This simplifies the design of the processors and Ensembles, which improves efficiency.

8.5 Related Work

Contemporary microprocessors rely extensively on hardware cache hierarchies to reduce apparent memory access latencies [133]. Processor designs commonly use cache block sizes that are larger than the machine word size, and transfer an entire cache block on a miss to provide a limited amount of hardware prefetching within the block.

Superscalar processors use out-of-order execution and non-blocking caches to hide memory access latencies by overlapping memory operations and with the execution of independent instructions [6, 79]. Register renaming is used to expose additional instruction-level parallelism, and allows processors to initiate more concurrent memory operations [152]. Speculative execution allows processors to calculate addresses and initiate cache refills early [152], which helps to reduce apparent memory access times. In addition to the structures used to track pending misses in the caches, superscalar processors use registers in the multi-ported renamed register file to receive data returned by load operations, and structures such as the instruction window and the memory reorder queue to record the state of pending memory operations. These structures are very expensive, which limits the extent to which superscalar processors can use out-of-order execution to hide memory access latencies.

Many high-performance processors use dedicated hardware prefetch units to speculatively fetch blocks that are expected to be accessed before the instructions that reference the blocks reach the memory system [72]. Because most attempt to predict future misses based on observed address miss patterns, hardware prefetch units are more effective when data and instruction access patterns exhibit significant amounts of structure and regularity. Consequently, they typically fail to comprehend complex access patterns and may fetch data that is never referenced before being released from the cache. Miss-based hardware prefetch schemes are typically only effective when data access patterns generate long stable address sequences because they require one or more misses to detect the access pattern. Hardware prefetch schemes typically prefetch beyond the end of a sequence. Dedicated stream buffers may be used to avoid polluting the data cache when the prefetched data is not accessed [72], though the misprediction wastes energy and memory system bandwidth.

Most miss-based hardware prefetch schemes, which typically resemble stream buffers [72], transfer linear sequences of cache blocks in response to observed cache miss patterns. These typically provide dedicated hardware to identify distinct reference streams and predict miss strides, and use the predicted strides to prefetch data within each stream. The TriMedia TM3270 media processor implements a region-based prefetch scheme that provides some software control over what data is prefetched when a load references an address within one of four software-defined memory regions [143]. The prefetch hardware allows software to configure the stride that is used to compute the prefetch address within the memory regions. This is useful when processing image data in scan-line order because the prefetch hardware can be configured to fetch data several rows ahead of the current processing position. The TriMedia processor transfers prefetch data directly to the data cache, rather than a dedicated stream buffer. The design relies on the associativity and capacity of the data cache to prevent the prefetched data from displacing active data and introducing an excessive number of additional conflict cache misses. A disadvantage of the TriMedia design, which is common to all

schemes that prefetch on accesses rather than on misses, is that a load operation that references an address within a prefetch region incurs two tag accesses and comparisons: one to resolve the load operation, and one to resolve whether the prefetch data is present in the cache or must be prefetched.

Software prefetching uses nonblocking prefetch instructions to initiate the migration of data to local caches before subsequent memory operations reference the data [20]. The technique is particularly effective in loops, as a loop body can often be constructed so that each iteration prefetches data that is accessed in a subsequent iteration. However, the prefetch instructions compete with other instructions for issue bandwidth, and the cache must be designed to handle multiple outstanding misses. Software prefetching may perform better than hardware prefetching when data access patterns lack regular structure and there is enough independent computation available to hide the memory access latency.

Vector processors can generate large numbers of independent parallel memory requests when applications exhibit structured memory access patterns [123]. Vector microprocessors typically use caches to capture temporal locality and reduce off-chip memory bandwidth demands [38]. The number of vector registers that are available to receive loads can limit the number of vector load operations that may be in flight, and consequently the extent to which vector memory operations can be used to hide memory access latencies. However, techniques such as vector register renaming [85] can be used to increase the number of physical registers and effective buffer capacity available for receiving loads.

The non-blocking cache designs required to satisfy the bandwidth demands of vector processors are complex, as they must support large numbers of concurrent misses to allow pending memory operations to saturate the available memory bandwidth. The Cray SV1 processor allows copies of vector data to be cached in a streaming cache chip. It uses a block size of a single word to avoid wasting memory bandwidth on non unit-stride accesses. The data caches required associative tag arrays capable of tracking hundreds of pending memory transactions [39]. Later designs used larger cache blocks to reduce the area and complexity of the hardware structures needed to track pending transactions [1].

Often, vector caches use dedicated structures such as replay queues to resolve references to cache lines that are in transient states when a memory operation is attempted [1]. Typically, additional dedicated data buffers are used to stage data returned from the memory system until it can be transferred to the vector cache and vector registers. These buffers are needed to decouple the memory system and processor, as it is difficult to design deeply pipelined memory systems that allow a requesting processor to stall the delivery of data. Because data buffer entries are reserved when requests are generated, the capacity of the data return buffers limits the amount of data that may be in flight. The traversal of these data buffers imposes additional energy costs on remote memory accesses.

Vector refill/access decoupling reduces the hardware complexity and cost of vector caches by using simpler hardware to fetch data before it is accessed, and by using the cache to buffer data returned from the memory system [17]. Vector memory operations are issued to a dedicated vector refill unit that attempts to fetch data that is missing from the cache before the vector memory operations are executed. This requires that the control processor be able to run ahead and queue vector memory and compute operations for the vector units, and relies on there being enough entries in the command queues to expose a sufficient number of pending vector commands to hide memory access latencies. The additional tag accesses performed by the

vector refill unit impose an additional energy cost on data cache accesses.

The Imagine stream processor [81] is unique in its use of stream load and store operations to transfer data between off-chip memory and its stream register file. It also provides stream send and receive operations that allow data to be transferred between the stream register file and remote devices connected by an interconnection network. The stream operations transfer records, and support non-unit stride, indexed, and other addressing modes that are specific to signal processing applications. The scoreboard used to enforce dependencies between memory operations and compute operations tracks stream transfers rather than individual record transfers, and allows computation to overlap with the loading and storing of independent streams. Elements of streams stored in the stream register file are always accessed sequentially through dedicated stream buffers. The stream buffers that connect the arithmetic clusters to the stream register file exploit the sequential stream access semantics and access 8 contiguous stream register locations in parallel; this allows the 8 SIMD arithmetic clusters to access the stream register file in parallel.

8.6 Chapter Summary

This chapter described the Elm memory system. Elm is designed to support many concurrent threads executing across an extensible fabric of processors. Elm exposes a hierarchical and distributed on-chip memory organization to software, and allows software to manage data placement and orchestrate communication. The organization allows the memory system to support a large number of concurrent memory operations. Elm exposes the memory hierarchy to software, and lets software control the placement and orchestrate the communication of data explicitly.

Chapter 9

Conclusion

9.1 Summary of Thesis and Contributions

This dissertation presented the Elm architecture and contemplated the concepts and insights that informed its design. The central motivating insight is that the efficiency at which instructions and data are delivered to function units ultimately dictates the efficiency of modern computer systems. The major contributions of this dissertation are a collection of mechanisms that improve the efficiency of programmable processors. These mechanisms are manifest in the Elm architecture, which provides a system for exploring and understanding how the different mechanisms interact.

Efficiency Analysis — This dissertation presented an analysis of the energy efficiency of a contemporary embedded RISC processor, and argued that inefficiencies inherent in conventional RISC architectures will prevent them from delivering the efficiencies demanded by contemporary and emerging embedded applications.

Instruction Registers — This dissertation introduced the concept of instruction registers. Instruction registers are implemented as efficient small register files that are located close to function units to improve the efficiency at which instructions are delivered.

Operand Registers — This dissertation described an efficient register organization that exposes distributed collections of operand registers to capture instruction-level producer-consumer locality at the inputs of function units. This organization exposes the distribution of registers to software, and allows software to orchestrate the movement of operands among function units.

Explicit Operand Forwarding — This dissertation described the use of explicit operand forwarding to improve the efficiency at which ephemeral operands are delivered to function units. Explicit operand forwarding uses the registers at the outputs of function units to deliver ephemeral data, which avoids the costs associated with mapping ephemeral data to entries in the register files.

Indexed Registers — This dissertation presented a register organization that exposes mechanisms for accessing registers indirectly through indexed registers. The organization allows software to capture reuse and locality in registers when data access patterns are well structured, which improves the efficiency at which data are delivered to function units.

Address Stream Registers — This dissertation presented a register organization that exposes hardware for computing structured address streams as address stream registers. Address stream registers improve efficiency by eliminating address computations from the instruction stream, and improve performance by allowing address computations to proceed in parallel with the execution of independent operations.

The Elm Ensemble Organization — This dissertation presented an Ensemble organization that allows software to use collections of coupled processors to exploit data-level and task-level parallelism efficiently. The Ensemble organization supports the concept of processor corps, collections of processors that execute a common instruction stream. This allows processors to pool their instruction registers, and thereby increase the aggregate register capacity that is available to the corps. The instruction register organization improves the effectiveness of the processor corps concept by allowing software to control the placement of shared and private instructions throughout the Ensemble.

This dissertation presented a detailed evaluation of the efficiency of distributing a common instruction stream to multiple processors. The central insight of the evaluation is that the energy expended transporting operations to multiple function units can exceed the energy expended accessing efficient local instruction stores. In exchange, the additional instruction capacity may allowing additional instruction working sets to be captured in the first level of the instruction delivery hierarchy. Consequently, the improvements in efficiency depend on the characteristics of the instruction working set. Elm allows software to exploit structured data-level parallelism when profitable by reducing the expense of forming and dissolving processor corps.

The Elm Memory System — This dissertation presented a memory system that distributes memory throughout the system to capture locality and exploit extensive parallelism within the memory system. The memory system provides distributed memories that may be configured to operate as hardware-managed caches or software-managed memory. This dissertation also described how simple mechanisms provided by the hardware allow the cache organization to be configured by software, which allows a virtual cache hierarchy to be adjusted to capture the sharing and reuse exhibited by different applications. The memory system exposes mechanisms that allow software to orchestrate the movement of data and instructions throughout the memory system. This dissertation described mechanisms that are integrated with the distributed memories to improve the handling of structured data movement within the memory system.

9.2 Future Work

There are many possible extensions and applications for the concepts and architectures presented in this dissertation.

Additional Applications — Mapping more applications to Elm, particularly system applications, would

provide additional insights into the performance and efficiency of the mechanisms and composition of mechanisms used in Elm. Many of the applications that we have used to evaluate Elm have regular data-level and task-level parallelism that map naturally to the architecture. It would be interesting to explore how less regular applications map to the architecture. A real-time graphics pipeline for a system such as a mobile phone handset would provide tremendous insight. Real-time graphics applications have significant data-level and task-level parallelism while imposing demanding load balance and synchronization challenges. Graphics applications also have interesting working sets and substantial memory bandwidth demands, which would provide additional insight into the performance of the memory system.

Extensions to Multiple Chips — It would be difficult to satisfy the computation demands of certain high-performance embedded systems using a single Elm chip. Many embedded systems, such as those used in automobiles, contain multiple processors distributed throughout the physical system. It would be interesting to explore system issues that arise when systems are constructed using multiple Elm processors. Ideally, it would be possible to connect multiple chips directly without requiring additional bridge chips, as this would allow small nodes to be constructed as a collection of Elm chips on a single board, and it would be interesting to explore how the memory system and interconnection network designs could be extended to support this efficiently. It would be interesting to explore whether the physical links that are used to connect to memory chips could be reused to establish links between Elm modules, so that pins could be flexibly assigned to memory channels or inter-processor links when the system is assembled. There would be interesting programming systems issues that would arise from mapping applications across multiple chips, as the bandwidth and latency characteristics of the links connecting the chips would differ significantly from the links connecting processors that are integrated on the same die.

Extend Arithmetic Operations — One way to extend the Elm architecture would be to provide support for additional arithmetic operations. The Elm architecture described in this dissertation was designed primarily to explore certain concepts and mechanisms, and we did not spend a significant amount of time considering details such as what arithmetic and logic operations should be supported. The efficiency of the architecture on many signal, image, and video processing applications could be improved by introducing sub-word SIMD operations. We specifically decided not to implement arithmetic instructions that operate on packed sub-word operands because the efficiency benefits of such extensions are well understood and are orthogonal to the central concepts that we intended to explore and that informed the design of architecture.

As the realized performance of modern high-performance computer systems is limited by power dissipation and cooling considerations, an interesting direction for continued research would be to explore how the concepts in Elm could be used in high-performance computer systems. This would obviously entail supporting floating point arithmetic operations, and would require that the processor architecture be extended to tolerate function units with longer operation latencies.

Efficient Circuits — Custom circuits techniques could be used to provide additional efficiency improvements. We designed the Elm architecture to allow most of the register files, forwarding networks, and communication fabrics to be realized using custom circuit design techniques without excessive design efforts. For example, low-voltage swing signaling techniques could be used to reduce the dynamic energy consumed

delivering operands and data to the processors and function units. It would be interesting to quantify the advantages of using aggressive custom circuits, and to compare the advantages of an optimized implementation to that of a system that supports instruction set customization within a conventional automated design flow.

Appendix A

Experimental Methodology

This supplementary chapter describes the experimental methodology and simulation infrastructure that we used to produce the data presented in the dissertation.

A.1 Performance Modeling and Estimation

The various performance data presented in this dissertation were collected from cycle-accurate and microarchitecture-precise simulators. We developed a configurable processor and system simulator for the Elm architecture described in this dissertation. The processor model implemented in the simulator is cycle-accurate and microarchitecture-precise, and allows for detailed data collection. The simulator model accepts a machine description file that allows parameters such as the instructions that may be executed by different function units, the number and organization of registers, and the capacity of different memories to be bound when a simulation loads. The system model similarly allows the number and organization of processors and various memories to be configured when a simulation loads. We developed a collection of test codes to test the functional correctness of the simulator and related simulation tools.

The simulator remains the primary software development infrastructure, and supports various features that are useful for testing and debugging applications, such as the ability to set breakpoints based on the contents of registers and memory locations, and the ability to dump various traces to files. The simulator implements a simple symbolic debugger that can be used to inspect and modify the state of the system as a simulation proceeds. To simplify debugging and assist with the development of regression tests, the simulator allows object code to specify symbolic information that may be used to refer to processors and memory locations. The simulator also supports features, such as the ability for any processor to handle system calls, that simplify the testing of application code, but would not be supported in most Elm systems.

We developed complete register-transfer level (RTL) models of the primary modules from which an Elm system is assembled. These include an Elm Ensemble, which includes a collection of 4 processors and shared memory, and a distributed memory module. These were used to validate the correctness and accuracy of the simulator models by running collections of test codes and kernels on both the simulator model and

the hardware model. The hardware model was tested extensively, with the intention that it be of sufficient quality to allow us to fabricate a test chip. The simulator was instrumented to provide detailed information about which instructions were executed, which registers accessed and bypassed, and so forth. We used the data provided by the instrumentation to assess the extent to which the collection of test codes exercised the hardware model. The RTL models are implemented in Verilog. The models implement fixed hardware configurations, and provide limited support for debugging and data collection.

The application data presented in this thesis was collected from compiled code. We developed a compiler and a programming system for Elm. All of the data reported in the dissertation are for codes compiled using these tools. The compiler and related tools are described in Chapter 4.

The kernel and application data reported corresponds to steady state behavior. Most of the kernels and applications presented in this dissertation process streams of data and have a few small critical instruction working sets. Consequently, the applications exhibit stable instruction working sets. We always wait until the instruction working sets are stable before beginning measurement; typically, this is achieved by waiting until the first few elements of the data stream have been processed before enabling measurement capture, though it sometimes involves executing applications with complicated control-flow multiple times. For most applications, we are able to fetch the resident data working sets into the local data memory and caches before beginning measurement. Consequently, most data cache misses and remote memory accesses correspond to compulsory misses, which occur when an application first accesses an element of a data stream.

A.2 Power and Energy Modeling

The reported power and energy data are for circuits implemented in a commercial 45 nm CMOS process that is intended for low standby-power applications, such as mobile telephone handsets. The process provides transistors with gate oxides and relatively high threshold voltages reduce leakage, and uses a nominal supply voltage of 1.1 V to compensate for the high threshold voltages.

With the exception of array circuits, we used synthesized RTL models of hardware to estimate power and energy data. The hardware is described in Verilog and VHDL. We used Synopsis Design Compiler to synthesize RTL models, and to map the models to an extensive standard cell library provided by the foundry. The RTL models use clock gating extensively throughout, and we used the automatic clock gating capabilities provided by Design Compiler to capture most of the remaining clocked elements.

We used Cadence Silicon Encounter to map the gate-level models to designs that are ready for fabrication. We used a design flow that follows that recommended by the foundry. We imported a floor plan that describes were modules should be placed, and fixed the placement of any hard macros such as memories. We used Encounter to construct the power distribution grid, and placed isolation fences and routing blockages around memories. We used Encounter to automatically place the standard cells, and then synthesized a clock tree. We then used Encounter to route and optimize the design. Encounter removes most of the inverters and buffers from the gate-level model produced by Design Compiler; the optimization phase inserts inverters and buffers, restructures logic, and resizes gates to improve timing. We closed timing using the fast library models for hold time constraints and the slow library models for setup time constraints. After inserting metal fill and fixing

design rule violations, we used Encounter to extract interconnect capacitances from the design, and output a gate-level model and accompanying standard delay format file that includes delays due to interconnect loads.

We used MentorGraphics ModelSim to simulate the execution of application code on the gate-level model produced by Encounter. We imported the delay file into the gate-level simulation to more accurately model signal propagation delays. We captured a value change dump (VCD) file during the simulation, which we subsequently imported into Encounter to estimate power. The VCD file captures the activity at every node in the gate-level simulation. We waited until an application reaches steady state before beginning the VCD capture.

To estimate the energy consumed executing some block of code, we imported the VCD file captured during its execution into Encounter and used Encounter to compute an estimate of the average power consumption. Encounter uses the VCD file to compute an average toggle rate for each node in the design. It uses the extracted interconnect capacitances and detailed activity-based power models provided in the standard cell library, which are derived from transistor-level simulations, to estimate power consumption. The estimated power consumption includes leakage power. We used the typical process library models, assume an operating temperature of 55°C, and a nominal supply voltage of 1.1 V to estimate leakage and dynamic power. Encounter provides a detailed accounting of the power consumed by each standard cell in the design, which includes the power consumed driving interconnect connected to the outputs a cell. We tabulated these to calculate the power consumed within specific modules within a design based on the module hierarchy extracted from the gate-level model. Encounter inserts some cells at the top of the module hierarchy when fixing design rule violations; we were generally able to infer which modules these are associated with based on the cell and net names. Finally, we integrated the average power to estimate the energy consumed executing a kernel.

We used a similar procedure to estimate the energy consumed performing basic logic and arithmetic operations, such as adding two 32-bit numbers. The procedure differed in that we simulated a single hardware module performing a sequence of operations in isolation. We used the resulting estimate of average power from Encounter to calculate the average energy expended performing each operation.

We used a different methodology to estimate the energy consumed accessing array circuits, such as register files and memories. These circuits exhibit significant structure and regularity, and typical implementations use custom designed circuits extensively. When adequate models are available from the foundry, we simply used the detailed energy models provided with the models. For example, the memories used to implement the caches for the processor discussed in Chapter 3 were provided by the foundry. When models are not available, we used custom energy models. These models are based on HSPICE simulations of circuits that we designed. We implemented the energy models as a collection of software tools that accept parameterized hardware descriptions and calculate estimates for the energy consumed accessing memory arrays, register files, and caches. We validated the software models by comparing estimates to HSPICE simulations of specific configurations. The HSPICE simulations include parasitic transistor and interconnect capacitances. We used Cadence Virtuoso to input layouts, and Synopsis StarRC to extract parasitic and interconnect capacitances. The layouts include 50% metal fill above the circuit being modeled. We used HSPICE to simulate

the transistor-level models extracted from layout, and measure the current drawn from the supply to estimate the energy consumed performing different operations. The circuit models include estimates of parasitic capacitances that are extracted from the layouts.

Appendix B

Descriptions of Kernels and Benchmarks

This supplementary chapter describes the kernels and benchmark codes that were used to evaluate the concepts and implementations described in the dissertation.

B.1 Description of Kernels

This section presents kernels that are used to study the performance and efficiency of different processor architectures discussed in the dissertation. The kernels are implemented in a variant of the C programming language that supports extensions for the Elk programming language.

aes

The advanced encryption standard (AES) is the common name of the Rijndael [31] symmetric-key cryptographic algorithm that was selected by the National Institute of Standards and Technology to protect sensitive information in an effort to develop a Federal Information Processing Standard [109]. The Rijndael algorithm is an iterated block cipher; it specifies a transformation that is applied iteratively on the data block to be encrypted or decrypted. Each iteration is referred to as a round, and the transformation a round function. Rijndael also specifies an algorithm for generating a series of subkeys from a key provided by the user at the start of the algorithm. Rijndael uses a substitution-linear transformation network with 10, 12, or 14 rounds; the number of rounds depends on the key size. The **aes** kernel uses 10 rounds and a 128-bit key.

Each of the cryptographic operations operates on bytes, and the data to be processed is partitioned into an array of bytes before the algorithm is applied. The round function consists of four layers. The first layer is a substitution layer that applies a substitution function to each byte. The second and third layers are linear mixing layers, in which the rows of the array are shifted and the columns are mixed. The fourth layer applies an exclusive-or operation to combine subkey bytes into the array. The column mixing operation is skipped in the last round.

conv2d

Two-dimension convolution is a common image and signal processing, and in computer vision applications. It is used to filter images, and is used in edge and feature detection algorithms. The structure of the data access patterns and computation are similar to those used to compute disparity maps from which depth information may be inferred in machine vision applications, and to perform motion vector searches in digital video applications.

The **conv2d** kernel applies the filter kernels used by the Sobel operator, which is used within edge detection algorithms in image processing applications [101]. The Sobel operator is a discrete differentiation operator that computes an approximation of the gradient of an image intensity function. It is based on convolving the image with a small, separable integer-valued filter in the horizontal and vertical directions, and the arithmetic operations required are therefore inexpensive to compute. The operator uses two 3×3 filter kernels, which are convolved with an image to compute approximations to the horizontal and vertical derivatives.

crc

Cyclic redundancy checks (CRCs) are hash functions designed to detect modifications of data, are particularly well suited to detecting error bursts, and are commonly used in digital communication networks and data storage systems to detect data corruption. The computation of a CRC resembles polynomial long division, except that the polynomial coefficients are calculated according to the modulo arithmetic of a finite field. The input data is treated as the dividend, the generator polynomial as the divisor, and the remainder is retained as the result of the CRC calculation. The **crc** kernel uses the 32-bit CRC-32 polynomial recommended by the IEEE and used in the IEEE 802.3 standards defining aspects of the physical and data link layers of wired Ethernet.

crosscor

In signal processing applications, cross-correlation provides a measure of similarity between two signals. The computation of the cross-correlation of two sequences resembles the computation of the convolution of two sequences. It can be used to detect the presence of known signals as components of other more complex and possibly noisy signals, and has applications in passive radar systems. In signal detection applications, the cross-correlation is typically computed as a function of a displacement or lag that is applied to one of the signals.

The **crosscor** kernel computes the cross-correlation of two discrete time signals at 16 time lags using a window of 64 samples. The signals are represented as sequences of 16-bit real-valued samples. The kernel computes a set of 16 cross-correlation coefficients, one for each time lag, and produces a set after each sample period.

dct

The discrete cosine transform [7] is an orthogonal transform representation that is similar to the discrete Fourier transform (DFT), but assumes both periodicity and even symmetry. Because the definition of the discrete cosine transform (DCT) involves only real-valued coefficients, the transformation of real-valued signals can be computed efficiently without using complex arithmetic operations. The DCT is common in digital signal and image processing applications, and the type-2 DCT is often used when coding for compression. The type-2 DCT is used in many standardized lossy compression algorithms because it tends to concentrate signal information in low-frequency components, which allows high-frequency components to be coarsely quantized or discarded. For example, both the algorithms defined by the JPEG image compression and the MPEG video compression standards use discrete cosine transforms.

The **dct** kernel implements the type-2 two-dimension DCT used in JPEG image compression. The transform is applied to the luminance and chrominance components of 8×8 blocks of pixels, producing equivalent frequency domain representations. The transform data require more bits to represent than the input image components. The luminance and chrominance components of the input image pixels are represented as 8-bit samples; the transformed samples are represented as 16-bit fixed-point samples. The transform is implemented as a horizontal transform across the image columns followed by a vertical transform across the rows.

fir

Finite impulse response (FIR) filters are prevalent in digital signal processing applications. They are inherently stable and can be designed to be linear phase. Common applications include signal conditioning, in which a signal from an analog-to-digital converter is digitally filtered before subsequent signal processing is performed. The **fir** kernel implements a 32-tap real-valued filter with 16-bit filter coefficients.

huffman

Huffman coding is an entropy coding algorithm for lossless data compression that uses a variable-length binary code table to encode symbols [70]. The variable-length codes used in Huffman codes are selected so that the bit string representing one symbol is never a prefix of a bit string representing a different symbol. The Huffman algorithm specifies how to construct a code table for a set of symbols given the expected frequency of the symbols to minimize the expected codeword length. Essentially, the code table is constructed so that more common symbols are always encoded using shorter bit strings than less common symbols. The term Huffman coding is often used to refer to prefix coding schemes regardless of whether the code is produced using Huffman's algorithm.

The **huffman** kernel implements the encoding scheme specified for the coding of quantized DC coefficients in the JPEG image compression standard. Rather than encode the DC coefficients directly, the scheme encodes the difference between the DC coefficient of successive transform blocks. The kernel computes and encodes the sequence of differences using code tables specified by the standard, assembling the resulting sequence of variable-length codes to form the coded bit-stream as symbols are coded.

jpeg

The JPEG standard defines a set of compression algorithms for photographic digital images. The lossy compression schemes are used in most digital cameras. Typically, JPEG encoding entails converting an image from RGB to YCbCr, spatially downsampling the chrominance components, splitting the resulting components into blocks of 8×8 components, converting the blocks to a frequency-domain representation using a discrete cosine transform, quantizing the resulting frequency components, and coding the quantized frequency components using a lossless coding scheme. The conversion from RGB to YCrCb and spatial downsampling of the chrominance components may not be performed. The quantization step provides most of the coding efficiency; it significantly reduces the amount of high frequency information that is encoded by coarsely quantizing high frequency components. The entropy coding scheme uses a run-length encoding scheme to encode sequences of zeros and an entropy coding scheme that is similar to Huffman coding to code non-zero symbols.

The **jpeg** kernel implements a JPEG encoder that encodes 24-bit RGB images and produces coded bit-streams. The **jpeg** kernel contains the DCT operation that is implemented in the **DCT** kernel, and the DC huffman coding operation that is implemented in the **huffman** kernel.

rgb2yuv

The YUV color space is used to represent images in common digital image and video processing applications because it accounts for human perception and allows less bandwidth to be allocated for chrominance components. Digital image and video compression schemes that are lossy and rely on human perceptual models to mask compression artifacts, such as those used in the JPEG and MPEG standards, typically operate on images that are represented in the YUV color space. Most commodity digital image sensors use a color filter array that selectively passes red, green, or blue light to selected pixel sensors, with missing color samples interpolated using a demosaicing algorithm. Consequently, digital cameras and digital video records that store images and video in a compressed format must convert image frames from RGB space to YUV space before encoding.

The **rgb2yuv** kernel implements the RGB to YUV conversion algorithm used in the MPEG digital video standards. The conversion uses 32-bit fixed-point arithmetic and 16-bit coefficients to map 8-bit interlaced RGB components to 8-bit interlaced YUV components. A positive bias of 128 is added to the U and V components to ensure they always assume positive values. The Y component is clamped to the range [0,235], and the U and V components are clamped to the range [0,240].

viterbi

The Viterbi algorithm [145] is a dynamic programming algorithm for finding the most likely sequence of hidden states that explains a sequence of observed events. It is used in many signal processing applications to decode bit-streams that have been encoded using forward error correction based on convolution codes. Viterbi decoders are common in digital cellular mobile telephone systems, wireless local area network modems,

and satellite communication systems. High-speed Viterbi detectors are used in magnetic disk drive read channels. The algorithm is also widely used in applications such as speech recognition, where hidden Markov models are used as language models, which construct sentences as sequences of words; word models, which represent words as sequences of phonemes; and acoustic models, which capture the evolution of acoustic signals through phonemes [83].

The algorithm operates in two phases. The first phase processes the input stream and computes branch and path metrics for paths traversing a trellis. Branch metrics are computed as normalized distances between the received symbols and symbols in the code alphabet, and represent the cost of traversing along a specific branch within a trellis. The path metric computation summarizes the branch metrics, and computes the minimum cost of arriving at a specific state in the trellis. The path metric computation is dominated by the add-compare-select operations that are used to compute the minimum cost of arriving at a state. After all of the symbols in an input frame have been processed, the second phase computes the likely sequence of transmitted data by tracing backwards

The **viterbi** kernel implements the decoder specified for the half-rate voice codec defined by the Global Standard for Mobile Communications (GSM), which is one of the most widely deployed standards for mobile telephone systems. The half-rate codec is used to compress 3.1 kHz audio to accommodate a 6.5 kbps transmission rate. The decoder has a constraint length of 5 and operates on a 189-bit frame. The input frame is represented as 3-bit soft-decision sample values, in which each input value encodes an estimate of the reliability of the corresponding symbol in the received bit-stream.

B.2 Benchmarks

The benchmarks used to evaluate Elm were implemented in Elk, a parallel programming language and runtime system we developed for mapping applications to Elm systems. The Elk programming language allows application developers to convey information about the parallelism and communication in applications, which allows the compiler to automatically parallelize applications and implement efficient communication schemes. Aspects of the Elk programming language syntax were inspired by StreamIt [141], and the following benchmarks were adapted from StreamIt implementations [49].

bitonic

Bitonic sort is a parallel sorting algorithm [16]. The **bitonic** benchmark implements a bitonic sorting network that sorts a set of N keys by performing $\log_2 N$ bitonic merge operations. The sort performs $O(N \log_2^2 N)$ comparisons.

dct

The **dct** benchmark implements a two-dimension inverse discrete cosine transform (IDCT) that complies with the IEEE DCT used in both the MPEG and JPEG standards. The IDCT is implemented as one-dimension

IDCT operations on the rows of the input data followed by one-dimension IDCT operations on the resulting columns. The application is structured to allow the compiler to map individual row and column IDCT operations to different processing elements.

des

The **des** benchmark implements the data encryption standard (DES) block cipher, which is based on a symmetric-key algorithm that uses a 56-bit key and operates on 64-bit blocks [25]. The application is structured to allow the initial and final permutation operations, and each of the operations comprising the 16 rounds to be mapped to different processing elements.

fmradio

The **fmradio** benchmark implements a software FM radio with a multi-band equalizer. The input signal stream is passed through a demodulator to recover the audio signal, which is then passed through an equalizer. The equalizer is implemented as a set of band-pass filters.

radar

Synthetic aperture radar combines a series of radar observations to produce an image of higher resolution than could be obtained from a single observation [130]. The **radar** benchmark implements an algorithm for reconstructing a synthetic aperture radar image using spatial frequency interpolation.

vocoder

The **vocoder** benchmark implements a voice bit-rate reduction coder for speech transmission systems that compresses the spectral envelope into low frequencies [129]. Mobile telephone systems use similar algorithms to reduce the channel bit-rates required to transmit voice signals with acceptable audio quality. All of the deconvolution, convolution, and filtering steps required by the scheme are performed in the frequency domain, and the coder applies an adaptive discrete Fourier transform (DFT) to successive windows of the input data to compute a high resolution spectrum of the speech.

Appendix C

Elm Instruction Set Architecture Reference

This supplementary chapter describes various aspects of the Elm instruction set architecture that reflect ideas that were presented in the dissertation. It does not provide a complete description of the instruction set and instruction set architecture.

C.1 Control Flow and Instruction Registers

This section describes aspects of the Elm instruction set architecture related to instruction registers. The Elm prototype execute statically scheduled pairs of instructions. The first instruction in the pair is executed in the ALU function unit, and the second instruction in the pair is executed in the XMU function unit. Control flow instructions and instruction register management instructions execute in the XMU function unit, and must be scheduled in the second instruction slot.

Control Flow Instructions

Control flow instructions transfer control within the instruction register file. Conditional control flow instructions such as branches are implemented using predicate registers to control whether the operation specified by a control flow instructions is performed.

Control flow instructions use instruction register names to identify the instruction to which control transfers. Names may be explicit, such as `ir31`. Explicit names may also be relative to the address of the control flow instructions, such as `ip+4`. This allows instructions within an instruction block to refer to each other without requiring that an instruction block be loaded at a fixed location in an instruction register file. Assembly code typically uses labels to specify relative targets of control flow instructions. The assembler recognizes labels that use the syntax `@label`, and converts them into explicit instruction register names when machine code is generated.

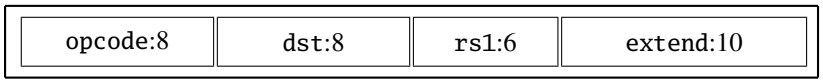


Figure C.1 – Encoding of Jump Instructions. The `dst` field encodes the destination, which specifies the instruction register to which control transfers. The destination may be specified as an absolute destination or a relative destination. Relative targets are resolved when the instruction is loaded into the instruction register.

Jump [`jmp`]

Jump instructions transfer control to the instruction stored in the instruction register specified as the destination. Conditional jumps are implemented using predicate registers to control whether the jump is performed. Control transfers when the predicate in the register specified by `rs1` agrees with the predicate condition.

```
jmp.always <destination>
jmp[.set|.clear] <rs1:pr> <destination:label>
<destination> ← <label>
              | ip + <offset:uint8>
              | <dst:ir>
```

The instruction format used to encode jump instructions is shown in Figure C.1. This instruction causes a pipeline stall in the instruction fetch (IF) stage when the destination instruction registers has a load pending. The jump instructions implemented in Elm have a single delay slot, and the instruction pair that follows the jump instruction is always executed regardless of whether the jump transfers control.

Loop [`loop`]

Loop instructions provide variants of jump instructions that expose hardware support for implementing loop constructs.

```
loop.always <destination:label>
loop[.set|.clear] <rs1:pr> <destination:label> (<count>)
<destination> ← <label>
              | ip + <offset:uint8>
              | <dst:ir>
<count>       ← <rs2>
              | <immediate:uint8>
```

Control transfers when the predicate register specified by `rs1` agrees with the predicate condition. If the predicate is set, indicating that the value zero is stored in the predicate register, the counter is loaded with the loop count encoded in the instruction; otherwise, the counter is decremented and the predicate updated. The counter is decremented in the decode stage; the counter is loaded in the write-back stage. The instruction encoded used for `loop` instructions is the same as the encoding used for `jmp` instructions and illustrated in Figure C.1.

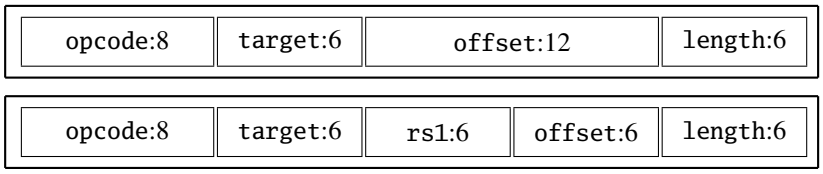


Figure C.2 – Encoding of Instruction Load Instructions. The `target` field specifies the instruction register at which the instruction block will be loaded. The target may be specified as an absolute position or relative to the instruction pointer. The address at which the instruction block resides in memory may specified as a fixed address encoded in the `offset` field or as an effective address computed by adding a short offset to the contents of `rs1`. The length of the instruction block is encoded in the `length` field.

Jump and Link [`jal`]

Jump and link operations allow software to implement procedure calls within the instruction registers. Jump and link operations transfer control to the instruction located at the instruction register specified as the destination and stores the value return instruction pointer address to the link register **`lr`**.

```
jal.always <destination:label>
jal[.set|.clear] <rs1:pr> <destination:label>
<destination> ← <label>
              | ip + <offset:uint8>
              | <dst:ir>
```

As with conventional jumps, conditional transfers of control are implemented using predicate registers. Control transfers and the link register is updated when the predicate in the register specified by `rs1` agrees with the predicate condition.

Jump Register [`jr`]

Jump register operations allow software to implement procedure returns within the instruction instructions. Jump register operations transfer control to the instruction in the instruction register specified the value stored in the link register **`lr`**.

```
jr.always <rs2:lr>
jr[.set|.clear] <rs1:pr> <rs2:lr>
```

As with conventional jumps, conditional transfers of control are implemented using predicate registers. Control transfers and the link register is updated when the predicate in the register specified by `rs1` agrees with the predicate condition.

Jump register operations may be used to implement transfers of control that specify destinations indirectly. For example, some switch statements may be implemented using jump register operations.

Predicate [`pred`]

Conditionally annuls the instruction that issued with the predicate instruction. Both the ALU and XMU function units can execute predicate instruction, and the predicate instruction may be paired with any other instruction.

```
pred[.set|.clear] <rs1:pr>
```

If the predicate value stored in the predicate register specified by `rs1` agrees with the predicate condition, the instruction that issues with the `pred` instruction is annulled in the decode stage.

Instruction Register Management

The following instructions allow software to load instruction blocks into instruction registers.

Instruction Load [`ild`]

Loads an instruction block to instruction registers.

```
ild [<destination>] [<address>] (<blocksize:uint6>)
<destination> ← <rd:ir>
              | ip + <offset:int>
<address>     ← <offset:int>
              | <rs1:ar> + <offset:int>
```

The instructions are loaded at the instruction register specified by the destination field, which may encode either an absolute or relative instruction register position. The location of the instruction block can be specified as an absolute address in memory, or it can be specified as a combination of an address register and offset. The instruction formats used to encode load instructions is illustrated in Figure C.2. The instruction load appears to take effect one cycle after the **`ild`** instruction executes.

Instruction Load and Jump [`jld`]

Loads an instruction block into registers and then jumps into the block. The operands and instruction formats are the same as an instruction load.

```
jld [<destination>] [<address>] (<blocksize:uint6>)
<destination>  ← <rd:ir>
               | ip + <offset:int>
<address>      ← <offset:int>
               | <rs1:ar> + <offset:int>
```

The jump instruction has a single delay slot, and the instruction load appears to take effect one cycle after the **`jld`** instruction executes. Consequently, the instruction pair that follows a **`jld`** instruction is always executed, and is read from the instruction registers before the first instructions from the instruction block loads.

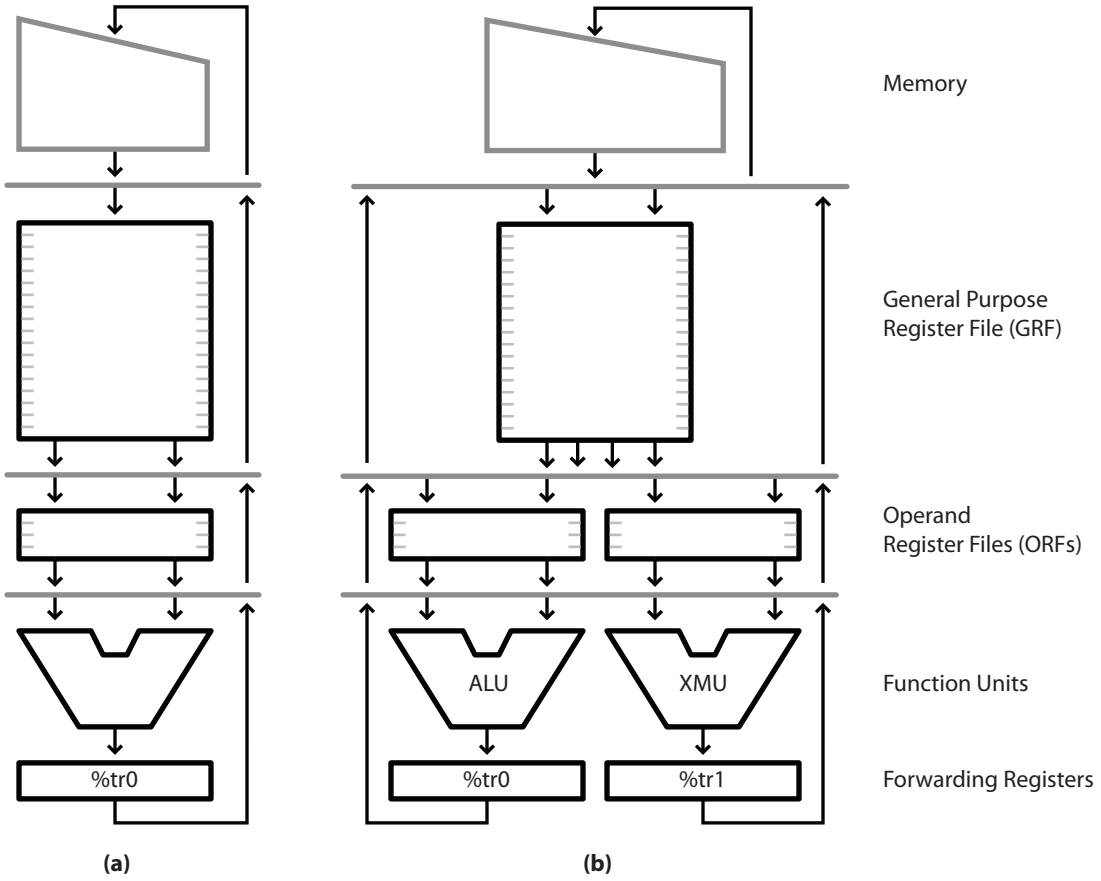


Figure C.3 – **Forwarding Registers.** The forwarding registers are located at the outputs of the function units.

C.2 Operand Registers and Explicit Operand Forwarding

This section discusses aspects of the instruction set architecture that are affected by explicit operand forwarding and operand registers. Both explicit operand forwarding and operand registers introduce additional architectural registers, and both affect how software schedules instructions.

Explicit Operand Forwarding

Explicit operand forwarding introduces forwarding registers to the instruction set architecture. The forwarding registers essentially expose the physical pipeline registers residing at the end of the execute (EX) stage of the pipeline. To expose additional opportunities for software to forward values through forwarding registers, the forwarding registers are preserved between instructions that update the pipeline registers, and are preserved when nop instructions execute. The forwarding registers are updated when an instruction departs the execute stage of the pipeline rather than when an instruction departs the write-back stage of the pipeline.

A forwarding register may be used as a source register in an instruction to specify that the instruction should read its operand directly from the corresponding pipeline register when the instruction reaches the execute stage of the pipeline. A forwarding register may be used as a destination register to specify that the result of the instruction should be discarded after departing the forwarding register. Effectively, the instruction is annulled in the write-back stage of the pipeline, and the result value is not driven back to the architectural register files.

In general, a forwarding register lets software access directly the result of the last instruction that the function unit associated with the forwarding register executed; the value becomes available when the instruction is in the write-back stage, before the instruction completes. However, forwarding registers behave differently when updated by a memory operation. The forwarding register associated with the function unit that executes memory operations receives the effective address when the operation executes, as this this is what is generated by the function unit in the execute stage of the pipeline. Thus, rather than storing the value read from memory when a load instruction executes, the forwarding register will store the address from which the load value was read. Furthermore, the forwarding register is updated both when store instructions and load instructions execute.

Because forwarding registers are associated with function units, software must know where an instruction will execute to determine which forwarding registers will be updated when the instruction executes. Determining which forwarding registers is updated when a processor has a single function unit (Figure C.3a) is trivial: there is only one forwarding register, and the forwarded value of each instruction is written to the forwarding register. Determining which forwarding register is updated when a processor has multiple function units (Figure C.3b) may be more complicated. If each function unit executes an disjoint subset of the instructions types, the instruction type will determine which forwarding register receives the result of the instruction. For example, if the ALU is the only function unit that can execute multiply instructions, the result of a multiple instruction will always be available in the forwarding registers associated with the ALU. Similarly, the the XMU is the only function unit that can execute load and store instructions, the forwarded value generated by a load or store instruction will always be available in the forwarding register associated with the XMU.

The Elm instruction set architecture defines a statically scheduled superscalar processor. Software decides where each instruction executes, and consequently knows statically which forwarding register will be updated by each instruction when it executes.

The situation is more complicated when multiple function units can execute an instruction. For example, multiple function units may be able to perform simple arithmetic operations, allowing an `add` instruction to execute on several function units. It may be possible for software to determine statically where an instruction will execute because of the registers that are referenced by the instruction. For example, if one of the source operands is an operand register, the instruction will execute on the function unit associated with the operand register. Similarly, the appearance of a forwarding register as the destination register of an instruction will force the instruction to execute on the function unit associated with the forwarding register. In general, however, it will not be possible for software to determine statically which function unit will execute an instruction in a dynamic superscalar processor that allows instructions to be executed by more than one

function unit.

Operand Registers

Operand registers are distributed among a processor's function units, as depicted in Figure C.3. The distribution of the registers is exposed to software, and the registers in each of the operand register files use different names to distinguish the registers. The Elm instruction set architecture refers to the operand registers local to the arithmetic and logic unit as data registers, and the operand registers local to the load-store unit as address registers. The data registers are named `dr0` – `dr3`, and the address registers `ar0` – `ar3`. The register names are needed to distinguish the registers and reflect typical usage rather than an enforced usage or allocation policy; data registers may be used in address computations and address registers in non-address computations.

Communication between the operand registers and function units is limited. An operand register can only be read by its local function unit, not by a remote function unit. However, an operand register can be written by any function unit, which allows the function units to communicate through operand registers rather than more expensive general-purpose registers. The instruction set architecture exposes the distribution of operand registers to software, and software is responsible for ensuring that an instruction can access all of its operands. This affects register allocation and instruction scheduling, as software must ensure that instructions will be able to access their operands when they execute. An instruction that attempts to read operand registers assigned to a remote function unit is considered malformed, and the result of the instruction is undefined and may differ between implementations.

The centralized general-purpose registers back the operand registers, as depicted in Figure C.3. The Elm instruction set architecture refers to the general-purpose registers as general-purpose registers `gr0` – `gr31`. These registers can be accessed by all of the function units. Normally, we would expect the general-purpose register file would provide enough read ports to allow both instructions to read their operands from the general-purpose registers as the instruction pair passes through the decode stage of the pipeline. However, a significant fraction of instruction read one or more of their operands from operand registers, which reduces demand for operand bandwidth at the general-purpose register file. The bandwidth filtering provided by the operand registers allows the number of read ports at the general-purpose register file to be reduced without adversely affecting performance.

The reduction in the number of read ports at the general-purpose register file introduces a structural hazard in the decode stage of the pipeline. Consider, for example, a pair of instructions that execute together and attempt to read four operands from general-purpose registers. With only two read ports to the general-purpose register file, all four operands cannot be read in a single cycle. Instead, the instruction pair would need to spend two cycles in the decode stage, with two of the four operands read in each cycle. A hardware interlock could be implemented to detect such conditions.

Rather than providing hardware interlocks, the instruction set architecture implemented in the Elm prototype exposes the number of read ports directly to software. The instruction set architecture allows an instruction pair to read at most two operands from general-purpose registers. Hardware determines which operands are read from general-purpose registers and routes the register specifiers and operands accordingly. Software

is responsible for ensuring that an instruction pair accesses no more than two general-purpose registers, which affects instruction scheduling and register allocation.

Forwarding and Interlocks

The introduction of explicit forwarding allows one to consider eliminating conventional hardware forwarding and instead to rely exclusively on software forwarding. Effectively, software would be responsible for forwarding results back to the function units using the forwarding registers. The operand registers provide storage locations with a one-cycle define-use latency, and the general-purpose registers provide storage locations with a two-cycle define-use latency. However, relying exclusively on software to forward values complicates scheduling and increases the number of **nop** instructions that need to be scheduled to ensure that operands are available where subsequent instructions expect them. It eliminates the hardware that tracks dependencies, which saves energy, but increases code size and results in more instructions being issued from the instruction registers. The Elm architecture provides hardware forwarding and interlocks because the reduction in the energy consumed loading and issuing instructions exceeded the energy consumed in the additional data-path control logic.

C.3 Indexed Registers and Address-Stream Registers

This section describes aspects of the instruction set architecture that relate to the index registers and address-stream registers.

Index and Indexed Registers

Index registers **xp0** – **xp3** store pairs of pointers that let software access the indexed registers indirectly. Indexed registers **xr0** – **xr2047** extend the general-purpose registers; indexed registers **xr0** – **xr31** are the same physical registers as general-purpose registers **gr0** – **gr31**. The Elm instruction set architecture allows a processor to implement between 32 and 2048 indexed registers. Figure C.4 illustrates the format used to transfer 64-bit index register values between memory and the index registers.

Each index register provides independent read and write pointers and independent read and write update increments. When an instruction specifies an index register appears as a source register, the read pointer value stored in the index register is used to read the operand from the indexed registers; the read pointer is updated in the decode stage after the operand is read from the indexed registers. When an instruction specifies an index register as a destination register, the write pointer value stored in the index register specifies the index register to which the result is written; the write pointer is updated in the write-back stage after the result of the operation is written to the destination indexed register. Should an index register appear in multiple source operand positions within an instruction pair, the read pointer is only updated once. Similarly, should an index register appear in multiple destination register positions within an instruction pair, the write pointer is only updated once. Software assigns each index register a range of indexed registers, and the read and write pointers are automatically adjusted to iterate through the assigned range of registers. Dedicated hardware

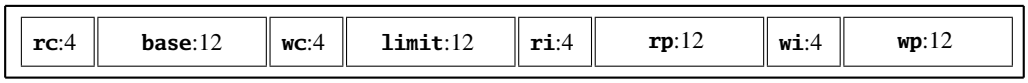


Figure C.4 – **Index Register Format.** The **base** and **limit** fields must be programmed so that repeatedly adding the read increment, decoded from the **ri** field, and write increment, decoded from the **wi** field, to the pointers will eventually cause the pointers to match **limit**.

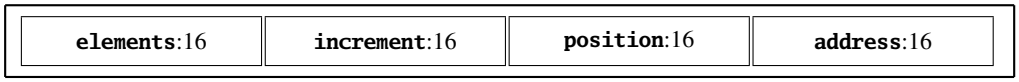


Figure C.5 – **Address-stream Register Format.** The **elements** field specifies the number of addresses in the stream. The **increment** field specifies the value to be added to **address** to generate the next address in the stream. The **position** field tracks the position in the stream. Both the **position** and **address** fields are set to zero when **position** reaches **elements**.

automatically updates a pointer after it is used by adding the value stored in the associated update increment field to the current pointer value; the hardware sets the pointer to its base value after it reaches its limit value.

Software can configure an index register to access the indexed registers as though the indexed registers were 16-bit registers rather than 32-bit registers. The read and write pointers within an index register are configured independently, which allows the index registers to be used to pack and extract pairs of 16-bit values using the 32-bit indexed registers. When an index register is configured to read the indexed registers as 16-bit registers, the index register can be configured to place the 16-bit value that is read from an indexed register in the upper or lower half of the 32-bit operand. The index register can also be configured to sign-extend or zero-extend a 16-bit value when it is read from an indexed register. When an index register is configured to write 16-bit values to the indexed registers, the index register can be configured to write either the upper 16 bits or lower 16 bits of a 32-bit result to the indexed registers.

Because the appearance of an index register in an instruction causes the indexed registers to be accessed, most instructions cannot access contents of an index register directly. Instead, the index register must be transferred to memory and modified there. The register value can be transferred between an index register and memory using the register save and restore operations described later.

Address-Stream Registers

The address-stream registers **as0** – **as3** establish a software interface to hardware in the XMU that accelerates the computation of fixed sequences of addresses. Each register has four fields that together capture the state of the address stream: **elements**, **increment**, **position**, and **address**. The **address** field holds the current address in the sequence. The **elements** field specifies the number of address elements in the sequence, and the **position** field holds the index of current stream in the sequence. The **increment** field specifies the value that is added to the value stored in the **address** field to compute the next address element in the sequence. Figure C.5 illustrates the format used to transfer 64-bit address-stream register values between memory and the address-stream registers.

Vector Memory Operations

Vector memory operations transfer multiple elements between indexed registers and memory. Vector memory operations are similar to convention memory operations except that each vector memory operation transfers multiple elements. Vector memory operations provide the same semantics as executing an equivalent sequence of scalar load and store instructions. The number of elements that a vector memory operation transfers is specified by software and encoded in the vector memory instruction. Address-stream registers can be used to generate the effective memory addresses at from which elements are loaded and to which elements are stored.

Vector Load — ld.v Vector load operations load multiple elements from memory to indexed registers. Vector load operations use the following assembly syntax.

```
ld.v <rd:xp> < address> ( <elements:uint6> )
<address> := [ <offset:int12> ]
           | [ <rs1> + <offset:int6> ]
           | [ <rs1> + <rs2> ]
```

The destination register specifies the index register that generates the sequence of indexed registers to which the elements are loaded. The value of `elements` encodes the number of elements that will be loaded. Except when an address-stream register appears in the address fields, the effective address computed from the contents of the source registers determines the address of the first element in the vector transfer. The address of subsequent elements is computed by incrementing the effective address. For example, the following instruction loads 4 elements from consecutive memory locations starting at effective address computed by adding 16 to the content of `ar1`.

```
ld.v xp0 [ar1+16] (4)
```

The following sequence of 4 scalar load operations produces the same effect as the vector load operation.

```
ld xp0 [ar1+16]
ld xp0 [ar1+17]
ld xp0 [ar1+18]
ld xp0 [ar1+19]
```

When an address-stream register is used to compute the effective address, the address-stream register is updated after each element is loaded and the effective address of the next element is computed using the updated address-stream register value. The address-stream registers allow vector memory operations to access memory with strides other than one. For example, the following vector memory operations loads 4 elements from the memory locations specified by the address sequence produced by adding the contents of `ar1` to the address stream produced by `as0`.

```
ld.v xp0 [ar1+as0] (4)
```

The following sequence of 4 scalar load operations produces the same effect as the vector load operation.

```
ld xp0 [ar1+as0]
ld xp0 [ar1+as0]
ld xp0 [ar1+as0]
ld xp0 [ar1+as0]
```

Vector Store — st.v Vector store operations store multiple elements from indexed registers to memory. Vector store operations use an assembly syntax similar to vector load operations.

```
st.v <rs:xp> < address> ( <elements:uint6> )
<address> := [ <offset:int12> ]
           | [ <rs1> + <offset:int6> ]
           | [ <rs1> + <rs2> ]
```

The source register specifies the index register that generates the sequence of indexed registers to be stored.

Though vector memory operations appear to execute as atomic operations, the memory operations that transfer individual elements may be interleaved with memory operations from other processors.

Saving and Restoring Index and Address-stream Registers

Usually, an instruction that references an index register or address-stream registers modifies the state of the register when it executes. The instruction set provides two operations, save and restore, to let software access registers such as the index registers and address-stream registers without modifying the registers and affecting the processor state.

In addition to allowing software to save and restore the register state of the processor, the save and restore operations are used to configure the index and address-stream registers. The save and restore operations are effectively distinguished store and load instructions that transfer 64-bit values between memory and registers. When used to store an index register or address-stream register, a save operation transfers the 64-bit value stored in the register to memory without affecting the state of the register. Software uses save operations to store the state of the index and address-stream registers when storing the state of a processor to memory. When used to load an index register or address-stream register, a restore operations transfers a 64-bit value from memory to the destination register. restore the register state of a processor from memory. Software uses restore instructions to initialize the index and address-stream registers, and to restore the state of index and address-stream registers when restoring the state of a processor from memory. The save and restore instructions are described below.

Save Register — `st.save` Saves register **`rd`** to the effective address computed when the instruction executes. The save operation is effectively a store operation that will transfer an entire register regardless of its size to memory without affecting the state of the saved register.

```
st.save <rd> <address>
<address> := [ <offset:uint18> ]
           | [ <rs1> + <offset:int12> ]
           | [ <rs1> + <rs2> ]
```

When register **rd** is a 32-bit register, the instruction stores the 32-bit value contained in **rd** to the effective memory address; when register **rd** is a 64-bit register, the save operation stores the 64-bit value contained in **rd** to memory. When register **rd** is an index register, the save operation stores the state of the index register to memory. The state of **rd** is not affected by the execution of the operation.

Restore Register — `ld.rest` Restores register **rd** from the effective address computed when the instruction executes. The restore operation is effectively a load operation that transfers an entire register regardless of its size from memory without otherwise affecting the state of the restored register.

```
ld.rest <rd> <address>
<address> := [ <offset:uint18> ]
           | [ <rs1> + <offset:int12> ]
           | [ <rs1> + <rs2> ]
```

When register **rd** is a 32-bit register, the operation loads the 32-bit stored at the effective address from memory to **rd**; when register **rd** is a 64-bit register, the operation loads the 64-bit value stored at the effective memory address to **rd**. When register **rd** is an index register, the restore operation loads the state of the index register from memory. The state of **rd** is updated to reflect the load operation, but is not subsequently modified.

C.4 Ensembles and Processor Corps

This section describes aspects of the Elm instruction set architecture that reflect the Ensemble organization of processors, memory, and interconnect.

Message Registers

Message registers `mr0` – `mr7` provide names for accessing the ingress and egress ports to the local communication fabric. A message register may be used where a general-purpose register is permitted as operand source and where a general-purpose register is permitted as a result destination. The synchronization bit associated with a message register is set to indicate the register is full when the message register is written, and it is set to indicate the register is empty when the register is read. The processor stalls when an instruction attempts to read a message register that is marked as empty until its synchronization bit indicates the register is full. The processor similarly stalls when an instruction attempts to write a message register that is marked as full until its synchronization bit indicates that the register is empty.

The synchronization bits associated with the message registers may be accessed through the processor control word in the processor status register. Software may inspect the synchronization bits to determine whether a message register is marked as full or empty, and may update the synchronization bits to mark message registers as full or empty without accessing the message registers. In some cases, software may prefer to inspect the synchronization bits to check whether a message register is empty before accessing it to avoid stalling the processor when the message register is empty. The synchronization bits may be transferred between memory and the processor status register using save and restore operations.

	Processor EP0	Processor EP1	Processor EP2	Processor EP3
mr0	EP0.mr0 → EP0.mr0	EP1.mr0 → EP0.mr1	EP2.mr0 → EP0.mr2	EP3.mr0 → EP0.mr3
mr1	EP0.mr1 → EP1.mr0	EP1.mr1 → EP1.mr1	EP2.mr1 → EP1.mr2	EP3.mr1 → EP1.mr3
mr2	EP0.mr2 → EP2.mr0	EP1.mr2 → EP2.mr1	EP2.mr2 → EP2.mr2	EP3.mr2 → EP2.mr3
mr3	EP0.mr3 → EP3.mr0	EP1.mr3 → EP3.mr1	EP2.mr3 → EP3.mr2	EP3.mr3 → EP3.mr3

Table C.1 – Mapping of Message Registers to Communication Links.

Message registers mr0 – mr3 are mapped to permanent connections between the processors within an Ensemble. The connectivity is shown in Table C.1. Each row of the table specifies the connection used to transfer data that are written to the specified message register. The message registers are mapped so that data written to message register mrj is transferred to processor Pj, and data arriving from processor Pk is received in message register mrk.

Messages registers mr4 – mr7 are mapped to configurable connections in the local communication fabric. These message registers may be mapped to configurable links in the local communication fabric that allows processors in different Ensembles to transfer data through the local communication fabric.

Local Communication and Synchronization Operations

The following instructions implement communication and synchronization operations.

Send [send]

Transfers a word from the source register rs1 to the message register rd. The destination message register is written in the write-back stage (WB) of the pipeline even when the synchronization bit assocaited with the message register indicates that the register is full.

```
send <rd:mr> <rs1>
```

The synchronization bit assocaited with the message register is always updated to indicate the message register is full when the instruction departs the write-back stage of the pipeline.

Receive [recv]

Transfers a word from inbound message register rs1 to the destination register rd. The message register is read in the decode stage (DE) of the even when the synchronization bit assocaited with the message register indicates that the register is empty.

```
recv <rd> <rs1:mr>
```

The synchronization bit assocaited with the message register is always updated to indicate the message register is empty when the instruction departs the write-back stage of the pipeline.

Barrier [barrier]

The barrier instruction provides a barrier synchronization mechanism to processors within an Ensemble. The processors participating in the barrier may be specified as a list of processor identifiers, which is encoded as a bit-vector in the immediate field of the instruction, or may be specified as a bit-vector stored in a register operand.

```
barrier <group>
 <group> ← ( <processors-list:immediate> )
         |  <rs1>
```

A processor signals that it has reached a barrier when the barrier instruction enters the write-back stage (WB) and there are no outstanding memory transactions. This ensures that all preceding instructions have cleared the pipeline and all memory operations have committed. The processor stalls with the barrier instruction in the write-back stage until all other processors participating in the barrier similarly signal that they have reached the barrier. The barrier instruction is immediately retired, and the processor continues to execute subsequent instructions.

Load Barrier [ld.b]

The load barrier operation combines a barrier and a load operation to provide a synchronized load operation. This ensures that a set of processors perform the load at the same time, and may be used to ensure that accesses by different processors to shared memory locations are coordinated.

```
ld.b <rd> < address> [ <processor-list> ]
<address> ← [ <offset:int12> ]
           | [ <rs1> + <offset:int6> ]
           | [ <rs1> + <rs2> ]
```

The processor list names the processors participating in the barrier operation. The load barrier operation may be used to synchronized loads from different processors to the same memory location to ensure that memory is accessed once and the result broadcast to multiple processors.

Store Barrier[st.b]

The store barrier operation combines a barrier and a store operation to provide a synchronized load operation. This ensures that a set of processors perform the load at the same time, and may be used to ensure that accesses by different processors to shared memory locations are coordinated.

```
st.b <rd> < address> [ <processor-list> ]
<address> ← [ <offset:int12> ]
           | [ <rs1> + <offset:int6> ]
           | [ <rs1> + <rs2> ]
```

The processor list names the processors participating in the barrier operation. The store barrier operation should not be used to synchronize stores from multiple processors to the same memory location. Which

processor successfully stores its value to memory will depend on how an implementation resolves the write conflict.

Processor Corps and Instruction Register Pools

Processors in a processor corps execute a common instruction stream. The execution model differs sufficiently from conventional single-instruction multiple-data execution models to justify some attention. Instructions are fetched from the shared instruction register pool and are issued to all of the processors in the corps. The processors execute the instructions independently, and may access different registers and different non-contiguous memory locations. Because the shared instruction register pool is formed by aggregating the instruction registers of processors participating in the processor corps, the shared instruction registers are physically distributed with the processors. The shared instruction registers appear to be accessed as though there were 4 instruction register banks. Each processor is responsible for fetching and issuing those shared instructions that reside in its local instruction registers. This improves instruction fetch and issue locality, and reduces the number of control signals that pass between the processors. Dedicated hardware orchestrates the fetching and issuing of shared instructions so that the process is transparent to software. Each processor is responsible for loading any shared instructions that reside in its local instruction registers. This keeps instruction load bandwidth local to each instruction register and ensures that all instruction load operates originate locally, which simplifies the design of the instruction load unit. It also increases the effective instruction load bandwidth that is available when loading shared instructions, as the processors in a corps can concurrently load shared instructions in parallel. However, software must explicitly schedule instruction load operations that load shared instructions so that the correct processor performs the load.

The organization of the instruction registers in an instruction register pool is illustrated in Figure C.6. Because a shared instruction register pool is formed by aggregating multiple processors' instruction registers, the shared instruction registers are physically distributed among the processors in a processor corps. Instruction registers in an instruction register pool are addressed using extended instruction register identifiers. The extended identifiers are formed by concatenating the processor identifier and its local instruction register address, as illustrated in Figure C.7. This encoding maps instruction registers in an instruction register file to adjacent identifiers, and allows extended instruction register identifiers to encode instruction register locality by exposing the physical distribution of instruction registers.

Loading Shared Instructions

To allow all operations that access instruction register file to be performed locally, each processor in a corps is responsible for managing its local instruction registers. For example, processor EP1 is responsible for issuing instructions from shared instruction registers `sir64` – `sir127`, and is responsible for loading instructions to shared instruction registers `sir64` – `sir127`. Shared instruction registers are loaded using instruction load operations to load instruction registers locally. The shared instruction register identifier is determined by where the instruction load operation executes and the instruction register address specified by the operation. For example, an instruction load operation that transfers instruction to instruction registers `ir8` – `ir16` will

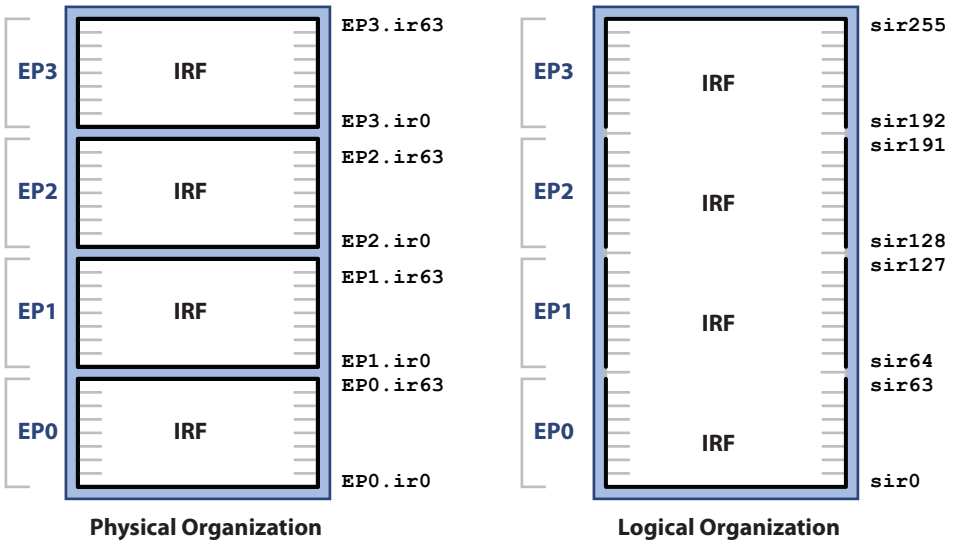


Figure C.6 – **Organization of the Ensemble Instruction Register Pool.** When combined to form a shared instruction register pool, the physical instruction registers are addressed so that sequential instructions are usually located in the same physical instruction register file. This organization keeps a significant fraction of the instruction fetch bandwidth local to the distributed instruction register files.

Figure C.7 – **Shared Instruction Register Identifier Encoding.** Shared instruction register identifiers encode an Ensemble processor identifier (`epid`) and a local identifier. The processor identifier specifies the instruction register file that contains the instruction register, and the local identifier specifies the address of the instruction register within an instruction register file.

load shared instruction registers `sir8` – `sir16` when executed by processor EP0. The same load operation will load instructions to shared instruction registers `sir72` – `sir80` when executed by processor EP1.

Predicate Registers and Processor Corps Flow Control

Processor corps execute the same standard set of flow control instructions as independent processors. The encodings used for flow control instructions allow shared instruction registers to be encoded as the target of flow control instructions. Conditional flow control instructions use predicate registers to control whether control transfers. When the state of a predicate register differs among the processors in a corps, a conditional flow control instruction will cause the processors in a corps to transfer control to different instructions. The target instruction register of a conditional flow control that may cause the processors in a corps to diverge must be a local instruction register.

Processor Corps Operations

The following instructions control the formation and dissolution of processor corps.

Jump as Processor Corps [`jmp.c`]

The jump as corps instruction allows processors in an Ensemble to form a processor corps. The instruction transfers control to the shared instruction register specified as the destination.

```
jmp.g <destination> [ <members> ]
<destination> ← <label>
              | <dst:sir>
```

The members field lists the identifiers of the processors that will form the corps. The processors begin executing as a corps after all of the processors listed in the instruction have executed the jump as corps instruction.

Jump as Independent Processors [`jmp.i`]

The jump as independent processor instruction allows processors in a processor corps to resume executing code as independent processors. The instruction transfers control to the instruction register specified as the destination.

```
jmp.i <destination>
<destination> ← <label>
              | <dst:ir>
```

The destination is local to each processor, and all of the processors in the corps will resume executing as independent processors at a common position in their local instruction registers. The processors in the corps decouple and begin executing as independent processors immediately after any processor in the corps executes a jump as independent processor instruction.

C.5 Memory System

This section describes the how the memory hierarchy is exposed through the instruction set architecture.

Address Space

Elm implements a shared address space that allows a processor to reference and access any memory location in the system. The address space is partitioned into four regions: local Ensemble memory, remote Ensemble memory, distributed memory, and system memory. We often refer to the local Ensemble memory as local memory and all other memory locations as remote memory, with the distinction being whether a memory operation escapes the local Ensemble. The partitioning is illustrated in Figure C.8. The capacity shown for each region reflects the maximum capacity based on the portion of the address space allocated to that

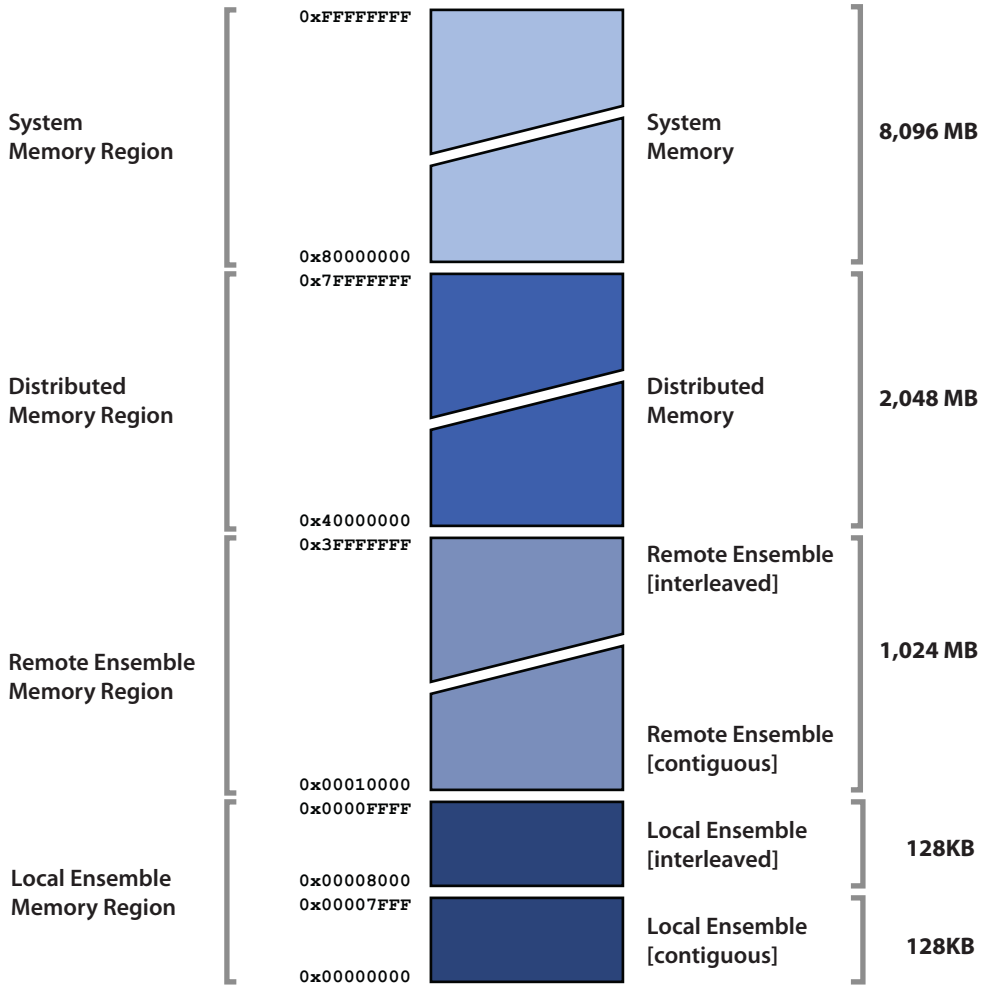


Figure C.8 – Address Space.

memory region, and does not necessarily reflect the provisioned capacity of the memories in any particular Elm system.

Local Ensemble Memory Region

The local Ensemble memory region provides a consistent mechanism for naming local Ensemble memory that is independent of where a thread is mapped in the system. Addresses in the local Ensemble address space always map to memory locations in the local Ensemble memory. This simplifies the generation of position independent code that will execute correctly regardless of where it executes. However, addresses in this range refer to different physical memory locations within different Ensembles, and addresses within the local Ensemble region have no meaning beyond of the Ensemble in which they are generated. Consequently, these addresses are commonly used for objects that are private and local to a thread, such as stacks and

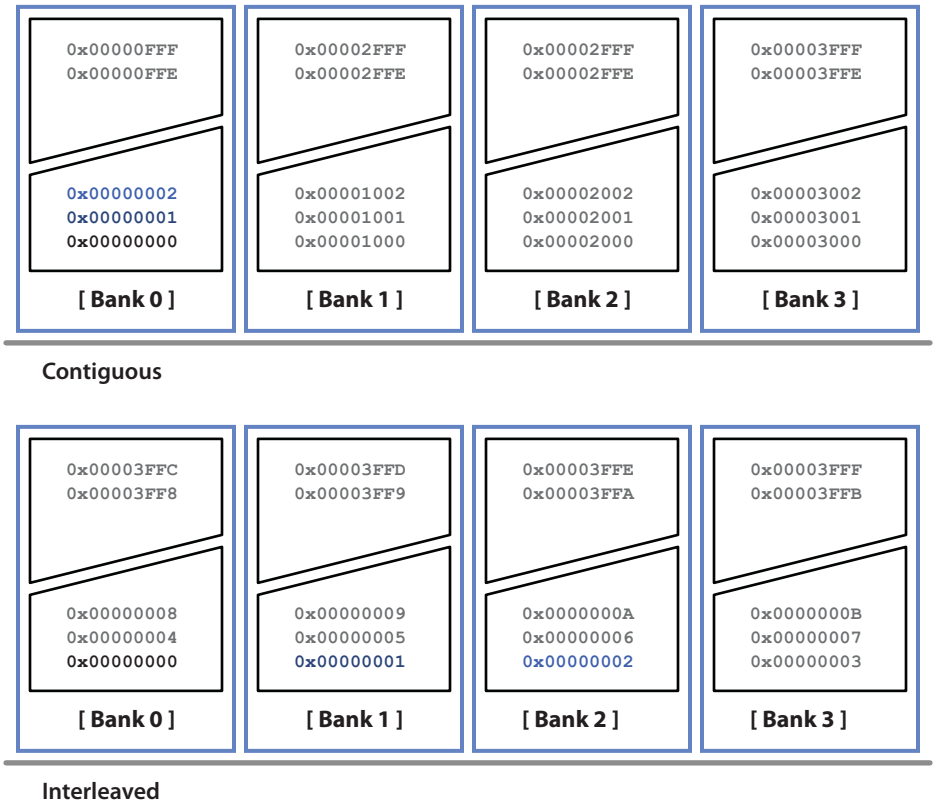


Figure C.9 – **Memory Interleaving.** Memory operations that transfer data between a contiguous region of memory and an interleaved region of memory can be used to implement conversions between structure-of-array and array-of-structure data object layouts.

temporaries.

Memory Interleaving

As shown in Figure C.8, each location in the local Ensemble memory appears twice in the local Ensemble address range: once with the memory banks contiguous, and once with the memory banks interleaved. In the contiguous address range, address bits in the middle of the address select the bank; the specific bits depend on the capacity of the Ensemble memory. In the interleaved address range, the least significant address bits select the bank. The difference is illustrated in Figure C.9. The contiguous and interleaved address ranges map to the same physical memory, so each physical memory word is addressed by two different local addresses.

The contiguous addressing scheme is useful when independent threads are mapped to an Ensemble. A thread may be assigned one of the local memory banks, and any data it accesses may be stored in its assigned memory bank. This avoids bank conflicts when threads within an Ensemble attempt to access the local memory concurrently, which improves performance and allows for more predictable execution. Because consecutive addresses map to the same memory bank, all of the data accessed by the thread resides within a

contiguous address range, and no special addressing is required to avoid bank conflicts.

The interleaved addressing scheme is useful when collaborative threads are mapped to an Ensemble. Adjacent addresses are mapped to different banks, and accesses to consecutive addresses are distributed across the banks. This allows data transfers between local Ensemble memory and remote memory to use consecutive addresses while distributing the data across multiple the banks, which improves bandwidth utilization when the memories are accessed by the processors within the Ensemble.

Remote Ensemble Memory Region

The remote Ensemble memory region comprises the aggregate of the Ensemble memory in a system. Addresses within this region have meaning throughout the system, and the remote Ensemble memory region effectively corresponds to the system view of all of the Ensemble memory. Addresses within the remote Ensemble address range have meaning both inside and outside of an Ensemble. The local Ensemble memory also appears in the remote Ensemble memory address range; where it appears depends on the global address assigned to the Ensemble. This convention provides a consistent mechanism for two Ensembles to name a memory location that resides within on of their Ensemble memories. Like the local Ensemble memory, each remote Ensemble memory appears twice: once in the continuous address range, and once in the interleaved address range. The interleaved addresses map to physical banks such that banks are interleaved within Ensembles but not across Ensembles, and the addresses assigned to an Ensemble in the interleaved and contiguous ranges are offset by a constant displacement.

Distributed Memory Region

The distributed memory is mapped into the distributed memory address range. Addresses within this range have meaning throughout the system, and are used to directly access memory locations in the Distributed Memories. When part of the distributed memory operates as a cache, memory accesses that attempt to access the distributed memory as though it were software-managed will access the cache control data instead. Loads return the tag and status information, and stores allow the tag and status information to be modified.

System Memory Region

The system memory region includes off-chip memory and memory mapped input-output regions. An Elm system may have multiple memory channels and independent memory controllers to provide additional parallelism within the memory system.

Software-Allocated Hardware-Managed Caches

Remote Memory Operations

The instruction set distinguishes between memory operations that access local memory and remote memory. The distinction allows software to indicate which memory operations require only local hardware resources,

and allows the hardware to optimize the handling of local memory operations. Both local and remote memory operations may access local memory. However, local memory operations may only access local memory, whereas remote memory operations may access any memory location. Remote memory operations use an on-chip interconnection network to transfer data between the Ensemble and remote memories. The network interface at the remote memory provides a serialization point. Remote memory operations that access the local Ensemble memory as a remote memory access the local memory through the network interface; this ensures that the operations are serialized with respect to other concurrent remote memory accesses.

Remote Load [ld.r]

Transfer data from memory to the destination register. The operation stalls in the execute (EX) stage if the network interface is unavailable. The operation always stalls in the write-back (WB) stage until the load value is received.

```
ld.r <rd:gr> <rs1> <rs2>
```

Remote Store [st.r]

Transfers data from a register to memory. The operation stalls in the execute (EX) stage if the network interface is unavailable. The operation stalls in the write-back (WB) stage until a message indicating that the store has completed is returned to the processor. This ensures that sequences of stores become visible to other processors in the order in which the stores execute.

```
st.r <rd:gr> <rs1> <rs2>
```

Cached Remote Load [ld.c]

The operation will stall in the execute (EX) stage if the network interface is unavailable. The operation always stalls in the write-back (WB) stage until the load value is received.

```
ld.c <rd:gr> <rs1> <rs2>
```

Cached Remote Store [st.c]

Remote cached store. The operation will stall in the execute (EX) stage if the network interface is unavailable. The operation stalls in the write-back (WB) stage until the message indicating that the store has completed is returned to the processor. This ensures that sequences of stores become visible to other processors in the order in which the stores execute.

```
st.c <rd:gr> <rs1> <rs2>
```

Decoupled Remote Load [ld.r.d]

Non-blocking remote load. A presence flag associated with the destination register, which must be a GR or XR, tracks when the load value arrives. Reading the register causes the processor to stall until the load value is received. Writing the register sets the presence flag and kills the load operation (the load value is discarded when it arrives).

ld.r.d <rd:gr> <rs1> <rs2>

Decoupled Remote Store [st.r.d]

Non-blocking remote store. Like a remote store, except the processor does not wait for the message indicating that the store has completed to be returned.

st.r.d <rd:gr> <rs1> <rs2>

Decoupled Cached Remote Load [ld.c.d]

Non-blocking cached remote load. A presence flag associated with the destination register, which must be a GR or XR, tracks when the load value arrives. Reading the register causes the processor to stall until the load value is received. Writing the register sets the presence flag and kills the load operation (the load value is discarded when it arrives).

ld.c.d <rd:gr> <rs1> <rs2>

Decoupled Cached Remote Load [st.c.d]

Non-blocking cached remote store. Like a remote store, except the processor does not wait for the message indicating that the store has completed to be returned.

st.c.d <rd:gr> <rs1> <rs2>

Atomic Compare-and-Swap [atomic.cas]

The address is specified by `rs1`, the comparison value by `rs2`, and the update value by `rs3`. The value at the memory location is returned to `rd`.

atomic.cas <rd> **[** <rs1> **]** <rs2> <rs3>

Atomic Fetch-and-Add [atomic.fad]

The memory location is specified by `rs1`. The current value at the memory location referenced by `rs1` is returned to `rd`. The value of register `rs2` is added to the contents of the memory location specified by `rs1`. If the sum is equal to the value of register `rs3`, the memory location is updated to 0. Otherwise, the memory location is updated to the sum.

```
atomic.fad <rd> [ <rs1> ] <rs2> <rs3>
```

Block Memory Operations

Block memory operations move contiguous blocks of memory through the memory system.

Block Get [get]

Transfers a block of memory from a remote memory location to a local memory location.

```
get [ <dst> ] [ <src> ] ( <size> )
```

Block Put [put]

Transfers a block of memory from a local memory location to a remote memory location.

```
put [ <dst> ] [ <src> ] ( <size> )
```

Block Move [move]

Transfers a block of memory from a remote memory location to a remote memory location.

```
move [ <dst> ] [ <src> ] ( <size> )
```

Stream Memory Operations

Stream memory operations transfer blocks of instructions and data through memory. A stream memory operation transfers a set of records, which are contiguous memory blocks, through the memory system.

Stream Descriptors

Streams are composed of sequences of fixed-length records. A stream descriptor describes the layout of a stream in memory, specifically where the records comprising the stream are located. Effectively, stream descriptors provide a compact representation with which software can efficiently describe a sequence of addresses to hardware. Hardware uses stream descriptors to calculate the addresses of the records when stream operations move records through the memory system. A stream descriptor provides the following information.

record-size — The size of each record in words.

stream-size — The number of records in the stream.

stride — The stride used to calculate the address of consecutive records.

index-address — The address at which an index vector resides. The entries of the index vector specify the offset of each record in the stream.

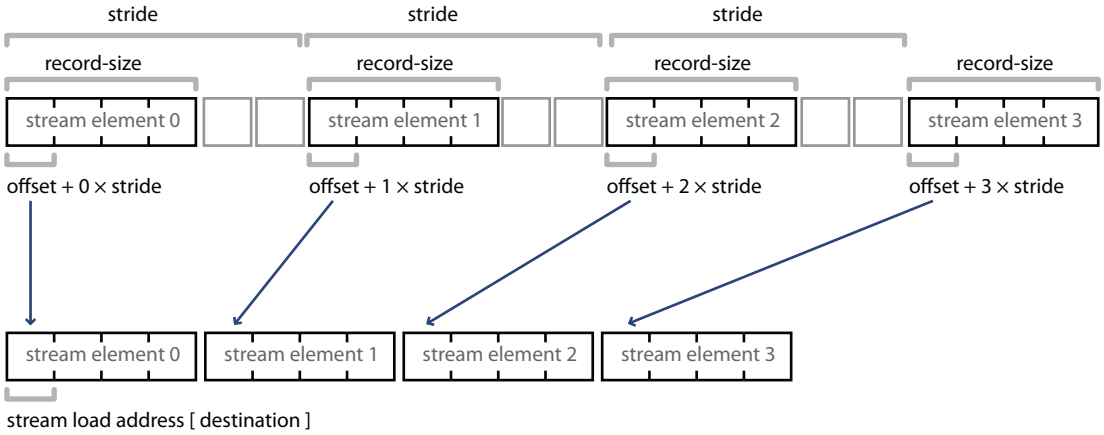


Figure C.10 – Strided Stream Addressing Load.

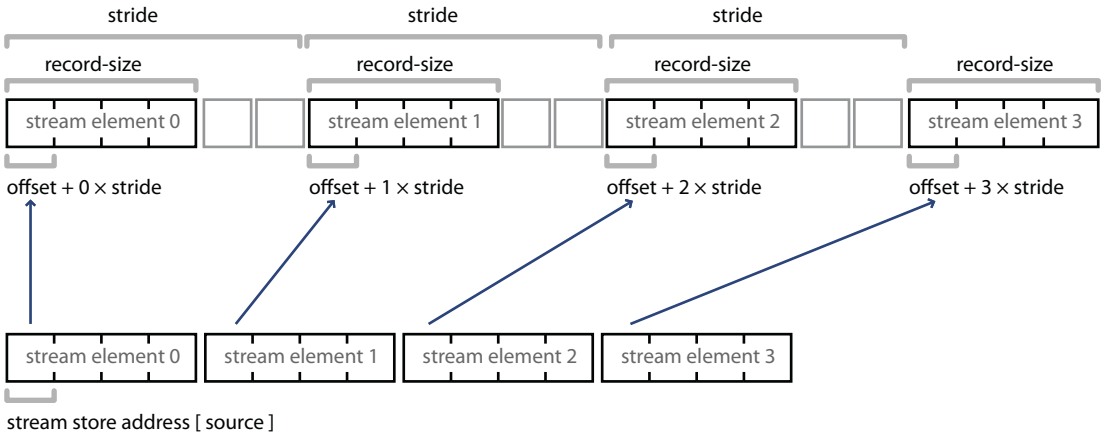


Figure C.11 – Strided Stream Addressing Store.

A stream memory operation uses a stream descriptor and an address offset to specify the addresses of records in the memory system; the offset is added to each address generated using the stream descriptor. Each stream operation specifies the number of records to be transferred and an address to which the stream is to be loaded or from which the records are to be stored. The SMA engine performing the stream operation maintains the state of the transfer so that a stream transfer can be performed as a sequence of smaller stream transfers. For example, a stream of N elements could be transferred by performing two stream transfers of $N/2$ elements. Figure C.10 and Figure C.11 illustrate how record addresses are generated when a stream memory operation uses strided addressing. Figure C.12 and Figure C.12 illustrate the use of the indexed addressing.

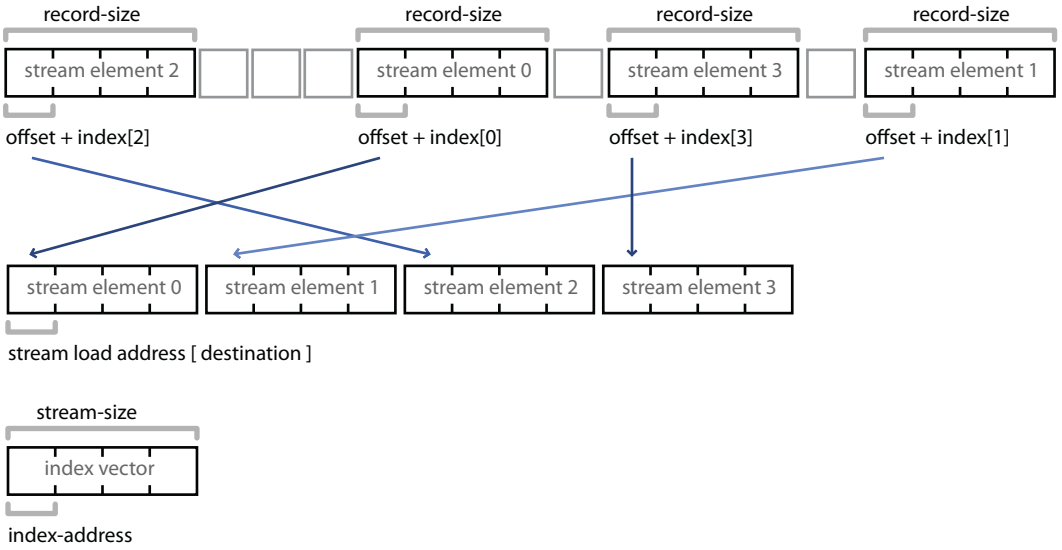


Figure C.12 – Indexed Stream Addressing Load.

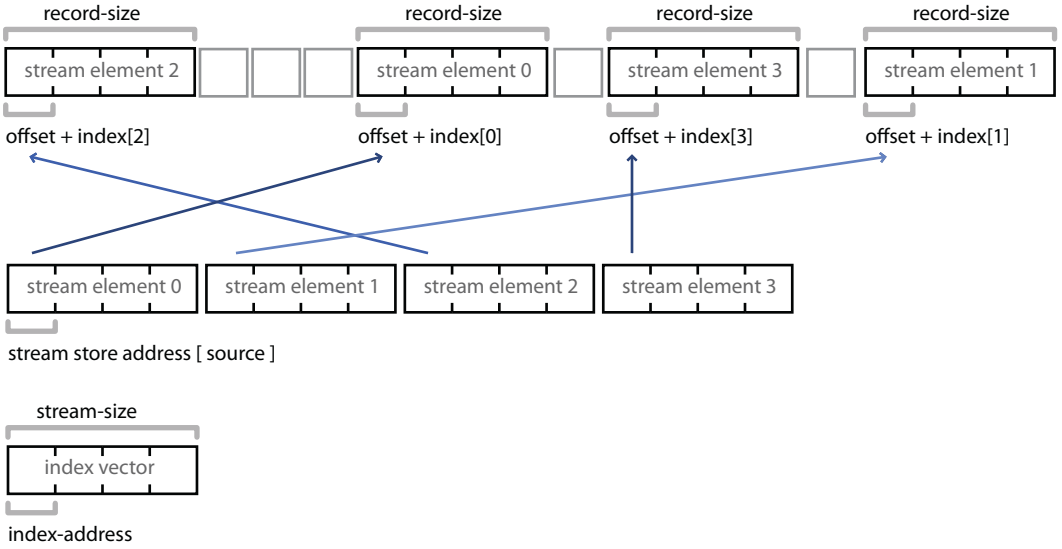


Figure C.13 – Indexed Stream Addressing Store.

Streaming Memory Transfers and Stream Descriptor Registers

Stream memory operations are bulk memory operations that move records through the memory system. The streaming memory access (SMA) engines accelerate stream memory operations by performing the address computations and memory operations needed to perform a stream memory operation. The engines are integrated with the memory controllers distributed through the memory system. The streams are described using stream descriptors. Each engine provides a small number of registers for storing stream descriptors. Software

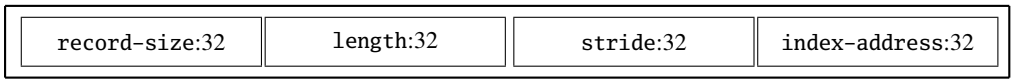


Figure C.14 – Record Stream Descriptor.

Figure C.15 – Instruction Format Used by Stream Operations.

registers a stream descriptor with an engine before initiating a stream memory operation.

Memory operations are performed over the on-chip network. The controllers exchange messages to complete memory operations. The following paragraphs describe the operations and the message formats. Each operation has a request and response message type. The request is used to initiate an operation, and the response to complete the operation or acknowledge that it completed.

Conceptually, streams are sequences of records. Stream memory operations move records through the memory hierarchy. Stream load operations collect and pack records into a contiguous range of memory, and stream store operations unpack and distribute records from a contiguous range of memory. Stream descriptors are used to describe the layout of the records in memory: the descriptors provide succinct descriptions that allow the hardware to determine where the records composing a stream reside in memory. The format of a stream descriptor is shown in Figure C.14.

The processor dispatches streaming memory operations to streaming memory access (SMA) engines, which are distributed throughout the memory system, instead of executing them directly. The operations are dispatched as messages that are sent over the on-chip network. The SMA engines coordinate the sequences of primitive memory operations (loads and stores) that are needed to complete a streaming memory operation. Address generators within the SMA engines generate the sequences of memory addresses. Streaming memory access engines are distributed throughout the memory system. To execute a streaming memory operation, a processor registers a stream descriptor with the SMA engine that will perform the operation and then dispatches the streaming memory operation to the SMA engine. The stream descriptors are composed in memory and then transferred to stream descriptor registers in SMA engines. The instruction format used by the stream memory operations is shown in Figure C.15.

Load Stream Descriptor [ld.sd]

This instruction transfers a stream descriptor from memory into a stream descriptor register in an SMA engine. The `address` argument provides the memory address, and the `register` address specifies the stream descriptor register.

```
ld.sd <register> <address>
```

```
  register ← [ <rd> ]
  addresss ← [ <rs1> + <rs2> ]
           | [ <rs1> + <immediate:int6> ]
```

Store Stream Descriptor [st.sd]

This instruction transfers a stream descriptor from a stream descriptor register in an SMA engine to memory. The `address` argument provides the memory address, and the `register` address specifies the stream descriptor register.

```
st.sd <register> <address>
  register ← [ <rd> ]
  addresss ← [ <rs1> + <rs2> ]
           | [ <rs1> + <immediate:int6> ]
```

Load Stream [ld.stream]

Loads stream elements to the address specified by `dest`. The remote addresses of the records are calculated by adding the `offset` operand to the sequence of addresses generated by the stream descriptor named by `stream`. The number of records to be transferred is specified by the `elements` operand.

```
ld.stream <dest> [ <stream> + <offset> ] ( <elements> )
  dest     ← [ <rd> ]
  stream   ← [ <rs1> ]
  offset   ← <rs2>
  elements ← <rs3>
           | <immediate:int6>
```

Store Stream [st.stream]

Stores stream elements from the address specified by `source`. The remote addresses of the records are calculated by adding the `offset` operand to the sequence of addresses generated by the stream descriptor named by by `stream`. The number of records to be transferred is specified by the `elements` operand.

```
st.stream <source> [ <stream> + <offset> ] ( <elements> )
  source   ← [ <rd> ]
  stream   ← [ <rs1> ]
  offset   ← <rs2>
  elements ← <rs3>
           | <immediate:int6>
```

Synchronize Stream Transfer [sync.stream]

Causes the processor to stall until a stream load or store completes. The operation is identified using the stream descriptor register, which is named by operand `rs1`.

```
sync.stream [ <rs1> ]
```

Bibliography

[1] D. Abts, S. Scott, and D.J. Lilja. So many states, so little time: Verifying memory coherence in the Cray X1. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[2] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *PODC '93: Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pages 159–170, New York, NY, USA, 1993. ACM.

[3] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT alewife machine: A large-scale distributed memory multiprocessor. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1991.

[4] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie, and Donald Yeung. The mit alewife machine: architecture and performance. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 1995. ACM.

[5] Anant Agarwal and Markus Levy. The kill rule for multicore. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 750–753, New York, NY, USA, 2007. ACM.

[6] T. Agerwala and J. Cocke. High Performance Reduced Instruction Set Processors. *IBM Thomas J. Watson Research Center Technical Report*, 558845, 1987.

[7] N. Ahmed, T. Natarajan, and K. R. Rao. Discrete cosine transfom. *IEEE Transactions on Computing*, 23(1):90–93, 1974.

[8] A. Allen, J. Desai, F. Verdico, F. Anderson, D. Mulvihill, and D. Krueger. Dynamic frequency-switching clock system on a quad-core itanium processor. In *Solid-State Circuits Conference - Digest of Technical Papers, 2009. ISSCC 2009. IEEE International*, pages 62 –63,63a, Feb. 2009.

[9] Arvind and Vinod Kathail. A multiple processor data flow machine that supports generalized procedures. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 291–302, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[10] Krste Asanović. *Vector Microprocessors*. PhD dissertation, University of California at Berkeley, Berkeley, CA, USA, 1998.

[11] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[12] James Balfour, William J. Dally, David Black-Schaffer, Vishal Parikh, and Jongsoo Park. An energy-efficient processor architecture for embedded systems. *Computer Architecture Letters*, pages 29–32, 2008.

[13] James Balfour, R. Curtis Harting, and William J. Dally. Operand registers and explicit operand forwarding. *Computer Architecture Letters*, 2009.

[14] Max Baron. Silicon à la carte. *Microprocessor Report*, January 2004.

[15] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 282–293, New York, NY, USA, 2000. ACM.

[16] K. E. Batcher. Sorting networks and their applications. In *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, New York, NY, USA, 1968. ACM.

[17] Christopher Batten, Ronny Krashinsky, Steve Gerding, and Krste Asanovic. Cache refill/access decoupling for vector machines. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 331–342, Washington, DC, USA, 2004. IEEE Computer Society.

[18] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, J. Liewei Bao, Brown, M. Mattina, C. Chyi-Chang Miao, Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. and Zook. TILE64 – Processor: A 64-Core SoC with Mesh Interconnect. pages 88 –598, Feb. 2008.

[19] David Black-Schaffer, James Balfour, William J. Dally, Vishal Parikh, and Jongsoo Park. Hierarchical instruction register organizations. *Computer Architecture Letters*, 2008.

[20] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. *ACM SIGARCH Computer Architecture News*, 19(2):52, 1991.

[21] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low Power CMOS Digital Design. *IEEE Journal of Solid State Circuits*, 27:473–484, 1995.

[22] L. Chang, D.J. Frank, R.K. Montoye, S.J. Koester, B.L. Ji, P.W. Coteus, R.H. Dennard, and W. Haensch. Practical strategies for power-efficient computing technologies. *Proceedings of the IEEE*, 98(2):215 –236, feb. 2010.

[23] Nathan Clark, Jason Blome, Michael Chu, Scott Mahlke, Stuart Biles, and Krisztian Flautner. An architecture framework for transparent instruction set customization in embedded processors. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 272–283, Washington, DC, USA, 2005. IEEE Computer Society.

[24] Robert P. Colwell, W. Eric Hall, Chandra S. Joshi, David B. Papworth, Paul K. Rodman, and James E. Tornes. Architecture and implementation of a VLIW supercomputer. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 910–919, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[25] D. Coppersmith. The Data Encryption Standard (DES) and its strength against attacks. *IBM Journal of Research and Development*, 38(3):243–250, 1994.

[26] IBM Corporation. *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*. IBM Corporation, February 2000.

[27] IBM Corporation. *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual*. IBM Corporation, August 2005.

[28] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual*. Intel Corporation, January 2006.

[29] Cray Research, Inc. *CRAY-1 Hardware Reference Manual, Rev. C*. Cray Resarch, Inc., 1977.

[30] José-Lorenzo Cruz, Antonio González, Mateo Valero, and Nigel P. Topham. Multiple-banked register file architectures. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 316–325, 2000.

[31] J. Daemen and V. Rijmen. *The design of Rijndael*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2002.

[32] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills. Architecture of a message-driven processor. In *ISCA '87: Proceedings of the 14th annual international symposium on Computer architecture*, pages 189–196, New York, NY, USA, 1987. ACM.

[33] William J. Dally, James Balfour, David Black-Schaffer, James Chen, R. Curtis Harting, Vishal Parikh, Jongsoo Park, and David Sheffield. Efficient embedded computing. *IEEE Computer*, July 2008.

[34] William J. Dally, J. A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Fyler. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, 1992.

[35] William J. Dally, Ujval J. Kapasi, Brucek Khailany, Jung Ho Ahn, and Abhishek Das. Stream processors: Progammability and efficiency. *Queue*, 2(1):52–62, 2004.

[36] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. Leblanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, pages 256–268, October 1974.

[37] Jeff H. Derby, Robert K. Montoye, and José Moreira. VICTORIA: VMX indirect compute technology oriented towards in-line acceleration. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 303–312, New York, NY, USA, 2006. ACM.

[38] Roger Espasa, Federico Ardanaz, Joel Emer, Stephen Felix, Julio Gago, Roger Gramunt, Isaac Hernandez, Toni Juan, Geoff Lowney, Matthew Mattina, and André Seznec. Tarantula: a vector extension to the alpha architecture. *SIGARCH Comput. Archit. News*, 30(2):281–292, 2002.

[39] Greg Faanes. A CMOS Vector Processor with a Custom Streaming Cache. In *Proceedings of Hot Chips*, volume 10, 1998.

[40] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic. The multicluster architecture: Reducing processor cycle time through partitioning. *International Journal of Parallel Programming*, 27(5):327–356, 1999.

[41] Kayvon Fatahalian and Mike Houston. A closer look at GPUs. *Commununications of the ACM*, 51(10):50–57, 2008.

[42] Joseph A. Fisher. Very long instruction word architectures and the eli-512. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*, pages 140–150, New York, NY, USA, 1983. ACM.

[43] Josh A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW approach to Architecture, Compilers and Tools*. Morgan Kaufmann Publishing, 2005.

[44] Mark Gebhart, Bertrand A. Maher, Katherine E. Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill, Stephen W. Keckler, Doug Burger, and Kathryn S. McKinley. An evaluation of the trips computer system. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 1–12, New York, NY, USA, 2009. ACM.

[45] Kanad Ghose and Milind B. Kamble. Reducing power in superscalar processor caches using sub-banking, multiple line buffers and bit-line segmentation. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pages 70–75, New York, NY, USA, 1999. ACM.

[46] Peter N. Glaskowsky. Cradle chip does anything. *Microprocessor Report*, October 1999.

[47] Ricardo Gonzalez, Benjamin M. Gordon, and Mark A. Horowitz. Supply and threshold voltage scaling for low power cmos. *IEEE Journal of solid-State Circuits*, 32:1210–1216, 1997.

[48] J. R. Goodman, Jian-tu Hsieh, Koujuch Liou, Andrew R. Pleszkun, P. B. Schechter, and Honesty C. Young. PIPE: a VLSI decoupled architecture. *SIGARCH Computer Architecture News*, 13(3):20–27, 1985.

[49] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 151–162, New York, NY, USA, 2006. ACM.

[50] A. Gordon-Ross, S. Cotterell, and F. Vahid. Tiny instruction caches for low power embedded systems. *Transactions on Embedded Computer Systems*, 2(4):449–481, 2003.

[51] S. Habnic and J. Gaisler. Status of the LEON2/3 processor developments. In *DASIA '07: DAta Systems In Aerospace 2007*. The Association of European Space Industry, 2007.

[52] Tom R. Halfhill. Picochip makes a big MAC. *Microprocessor Report*, 2003.

[53] Tom R. Halfhill. Tensillica tackles bottlenecks. *Microprocessor Report*, May 2004.

[54] Tom R. Halfhill. Ambric's new parallel processor. *Microprocessor Report*, October 2006.

[55] Tom R. Halfhill. Intel's tiny atom. *Microprocessor Report*, April 2009.

[56] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[57] Mark Jerome Hampton. *Exposing Datapath Elements to Reduce Microprocessor Energy Consumption*. SM dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 2001.

[58] Mark Jerome Hampton. *Reducing exception management overhead with software restart markers*. PhD dissertation, Massachusetts Institute of Technology, Cambridge, MA, USA, 2008.

[59] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*, pages F1–F53. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[60] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[61] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[62] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[63] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[64] S. Hines, G. Tyson, and D. Whalley. Reducing instruction fetch cost by packing instructions into register windows. In *Proceedings of ACM/IEEE International Symposium on Microarchitecture*, 2005.

[65] Stephen Hines, David Whalley, and Gary Tyson. Adapting compilation techniques to enhance the packing of instructions into registers. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 43–53, New York, NY, USA, 2006. ACM.

[66] Stephen Hines, David Whalley, and Gary Tyson. Guaranteeing hits to improve the efficiency of a small instruction cache. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 433–444, Washington, DC, USA, 2007. IEEE Computer Society.

[67] Ron Ho, Ken Mai, and Mark Horowitz. Managing wire scaling: A circuit perspective. In *Proceedings of the IEEE 2003 International Interconnect Technology Conference*, June 2003.

[68] Ron Ho, Kenneth W. Mai, Student Member, and Mark A. Horowitz. The future of wires. In *Proceedings of the IEEE*, pages 490–504, 2001.

[69] Mark Horowitz. Scaling, power and the future of cmos. In *VLSID '07: Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference*, page 23, Washington, DC, USA, 2007. IEEE Computer Society.

[70] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineering*, pages 1098–1101, September 1952.

[71] Q. Jacobson and J.E. Smith. Instruction pre-processing in trace processors. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 125. IEEE Computer Society Washington, DC, USA, 1999.

[72] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers. In *International Symposium on Computer Architecture*, pages 388–397. ACM New York, NY, USA, 1998.

[73] Ujval J. Kapasi, William J. Dally, Scott Rixner, Peter R. Mattson, John D. Owens, and Brucek Khailany. Efficient conditional operations for data-parallel architectures. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 159–170, New York, NY, USA, 2000. ACM.

[74] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, 2003.

[75] Michal Karczmarek, William Thies, and Saman Amarasinghe. Phased scheduling of stream programs. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 103–112, New York, NY, USA, 2003. ACM.

[76] Vinod Kathail, Michael S. Schlansker, and B. Ramakrishna Rau. HPL-PD architecture specification: Version 1.1. Technical report, February 1994.

[77] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 140–151, New York, NY, USA, 2009. ACM.

[78] John H. Kelm, Daniel R. Johnson, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. A task-centric memory model for scalable accelerator architectures. In *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 77–87, Washington, DC, USA, 2009. IEEE Computer Society.

[79] R.E. Kessler et al. The alpha 21264 microprocessor. *IEEE micro*, 19(2):24–36, 1999.

[80] B. Khailany, T. Williams, J. Lin, E. Long, M. Rygh, D. Tovey, and W.J. Daly. A programmable 512 gops stream processor for signal, image, and video processing. pages 272 –602, Feb. 2007.

[81] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.

[82] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 184–193, Washington, DC, USA, 1997. IEEE Computer Society.

[83] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

[84] Christoforos Kozyrakis and David Patterson. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 283–293, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[85] Christos Kozyrakis and David Patterson. Overcoming the limitations of conventional vector processors. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 399–409, New York, NY, USA, 2003. ACM.

[86] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating message-passing and shared-memory: early experience. *SIGPLAN Not.*, 28(7):54–63, 1993.

[87] Ronny Krashinsky, Christopher Batten, and Krste Asanović. Implementing the scale vector-thread processor. *ACM Trans. Des. Autom. Electron. Syst.*, 13(3):1–24, 2008.

[88] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanović. The vector-thread architecture. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 52, Washington, DC, USA, 2004. IEEE Computer Society.

[89] Ronny Meir Krashinsky. *Vector-thread architecture and implementation*. PhD thesis, Cambridge, MA, USA, 2007. Adviser-Asanovic, Krste.

[90] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization of multiprocessors with shared memory. *ACM Trans. Program. Lang. Syst.*, 10(4):579–601, 1988.

[91] John Kubiatowicz and Anant Agarwal. Anatomy of a message in the alewife multiprocessor. In *ICS '93: Proceedings of the 7th international conference on Supercomputing*, pages 195–206, New York, NY, USA, 1993. ACM.

[92] R. Kumar and G. Hinton. A family of 45nm ia processors. In *Solid-State Circuits Conference - Digest of Technical Papers, 2009. ISSCC 2009. IEEE International*, pages 58 –59, Feb. 2009.

[93] M. Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328, New York, NY, USA, 1988. ACM.

[94] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[95] Lea Hwang Lee, Bill Moyer, and John Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *ISLPED '99: Proceedings of the 1999 international symposium on Low power electronics and design*, pages 267–269, New York, NY, USA, 1999. ACM.

[96] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. *SIGPLAN Not.*, 33(11):46–57, 1998.

[97] A.S. Leon, J.L. Shin, K.W. Tam, W. Bryg, F. Schumacher, P. Kongetira, D. Weisner, and A. Strong. A power-efficient high-throughput 32-thread sparc processor. pages 295 – 304, Feb. 2006.

[98] Yuan Lin, Hyunseok Lee, Mark Woh, Yoav Harel, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. SODA: A low-power architecture for software radio. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 89–101, Washington, DC, USA, 2006. IEEE Computer Society.

[99] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[100] John D. C. Little. A proof of the queuing formula: $L = \lambda W$. *Operations Research*, 9(3):383–387, 1961.

[101] D. Marr and E. Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London. Series B, Biological Sciences*, pages 187–217, 1980.

[102] Yves Mathys and André Châtelain. Verification strategy for integration 3G baseband SoC. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 7–10, New York, NY, USA, 2003. ACM Press.

[103] Stephen Melvin and Yale Patt. Enhancing instruction scheduling with a block-structured ISA. *International Journal of Parallel Programming*, 23(3):221–243, 1995.

[104] Sun Microsystems. *UtraSPARC Architecture 2007*. Sun Microsystems, draft D0.9.3b edition, October 2009.

[105] J. Montanaro, R.T. Witek, K. Anne, A.J. Black, E.M. Cooper, D.W. Dobberpuhl, P.M. Donahue, J. Eno, W. Hoeppner, D. Kruckemyer, T.H. Lee, P.C.M. Lin, L. Madden, D. Murray, M.H. Pearce, S. Santhanam, K.J. Snyder, R. Stehpany, and S.C. Thierauf. A 160-MHz, 32-b, 0.5-w CMOS RISC microprocessor. *Solid-State Circuits, IEEE Journal of*, 31(11):1703–1714, Nov 1996.

[106] J. H. Moreno, V. Zyuban, U. Shvadron, F. D. Neeser, J. H. Derby, M. S. Ware, K. Kailas, A. Zaks, A. Geva, S. Ben-David, S. W. Asaad, T. W. Fox, D. Littrell, M. Biberstein, D. Naishlos, and H. Hunter. An innovative low-power high-performance programmable signal processor for digital communications. *IBM Journal of Reseasrch and Development*, 47(2-3):299–326, 2003.

[107] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A design space evaluation of grid processor architectures. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 40–51, Washington, DC, USA, 2001. IEEE Computer Society.

[108] U.G. Nawathe, M. Hassan, K.C. Yen, A. Kumar, A. Ramachandran, and D. Greenhill. Implementation of an 8-core, 64-thread, power-efficient sparc server on a chip. *Solid-State Circuits, IEEE Journal of*, 43(1):6 –20, Jan. 2008.

[109] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback. Report on the development of the Advanced Encryption Standard (AES). *JOURNAL OF RESEARCH-NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY*, 106(3):511–576, 2001.

[110] Richard van Nee and Ramjee Prasad. *OFDM for Wireless Multimedia Communications*. Artech House, Inc., Norwood, MA, USA, 2000.

[111] Chris J. Newburn, Andrew S. Huang, and John Paul Shen. Balancing fine- and medium-grained parallelism in scheduling loops for the ximd architecture. In *PACT '93: Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 39–52, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.

[112] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGPLAN Not.*, 31(9):2–11, 1996.

[113] Gregory M. Papadopoulos and David E. Culler. Monsoon: an explicit token-store architecture. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 82–91, New York, NY, USA, 1990. ACM.

[114] David A. Patterson and Carlo H. Sequin. Risc i: A reduced instruction set vlsi computer. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 443–457, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[115] M.J.M. Pelgrom, A.C.J. Duinmaijer, A.P.G. Welbers, et al. Matching properties of MOS transistors. *IEEE journal of solid-state circuits*, 24(5):1433–1439, 1989.

[116] Miquel Pericàs, Eduard Ayguadé, Javier Zalamea, Josep Llosa, and Mateo Valero. Power-efficient VLIW design using clustering and widening. *International Journal of Embedded Systems*, 3(3):141–149, 2008.

[117] L. Pileggi, G. Keskin, X. Li, K. Mai, and J. Proesel. Mismatch analysis and statistical design at 65 nm and below. In *IEEE Custom Integrated Circuits Conference*, 2008.

[118] B. Ramakrishna Rau, Michael S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 158–169, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[119] B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towie. The cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *Computer*, 22(1):12–26, 28–30, 32–35, 1989.

[120] Suzanne Rivoire, Rebecca Schultz, Tomofumi Okuda, and Christos Kozyrakis. Vector lane threading. *International Conference on Parallel Processing*, 0:55–64, 2006.

[121] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 3–13, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[122] Scott Rixner, William J. Dally, Brucek Khailany, Peter R. Mattson, Ujval J. Kapasi, and John D. Owens. Register organization for media processing. In *HPCA 6: Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 375–386, 2000.

[123] Richard M. Russell. The CRAY-1 computer system. *Commun. ACM*, 21(1):63–72, 1978.

[124] S. Rusu, S. Tam, H. Muljono, J. Stinson, D. Ayers, J. Chang, R. Varada, M. Ratta, and S. Kottapalli. A 45nm 8-core enterprise xeon processor. In *Solid-State Circuits Conference - Digest of Technical Papers, 2009. ISSCC 2009. IEEE International*, pages 56 –57, Feb. 2009.

[125] N . Sakran, M . Yuffe, M . Mehalel, J . Doweck, E . Knoll, and A . Kovacs. The implementation of the 65nm dual-core 64b Merom processor. In *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 106 –590, Feb. 2007.

[126] Michael S. Schlansker and B. Ramakrishna Rau. Epic: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, 2000.

[127] J. Schutz and C. Webb. A scalable x86 cpu design for 90 nm process. In *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International*, volume 1, pages 62 – 513, Feb. 2004.

[128] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[129] S. Seneff. System to independently modify excitation and/or spectrum of speech waveform without explicit pitch extraction. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 30(4):566 – 578, aug 1982.

[130] C. W. Sherwin, J. P. Ruina, and R. D. Rawcliffe. Some early developments in synthetic aperture radar systems. *Military Electronics, IRE Transactions on*, MIL-6(2):111 –115, april 1962.

[131] Olli Silven and Kari Jyrkkä. Observations on power-efficiency trends in mobile communication devices. *EURASIP Journal of Embedded Systems*, 2007(1):17–17, 2007.

[132] Mikhail Smelyanskiy, Gary S. Tyson, and Edward S. Davidson. Register queues: A new hardware/software approach to efficient software pipelining. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[133] A.J. Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3):473–530, 1982.

[134] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *SPIE Vol. 298 Real-Time Signal Processing IV*, pages 241–248. Denelcor, Inc., 1981.

[135] J. E. Smith, Greg Faanes, and Rabin Sugumar. Vector instruction set support for conditional operations. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 260–269, New York, NY, USA, 2000. ACM.

[136] Alex Solomatnikov, Amin Firoozshahian, Wajahat Qadeer, Ofer Shacham, Kyle Kelley, Zain Asgar, Megan Wachs, Rehan Hameed, and Mark Horowitz. Chip multi-processor generator. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 262–263, New York, NY, USA,, 2007. ACM.

[137] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[138] Michael Bedford Taylor, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Scalar operand networks: On-chip interconnect for ilp in partitioned architectures. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 341, Washington, DC, USA, 2003. IEEE Computer Society.

[139] Michael Bedford Taylor, Walter Lee, Jason Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul Johnson, Jason Kim, James Psota, Arvind Saraf, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. Evaluation of the Raw microprocessor: An

exposed-wire-delay architecture for ILP and streams. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 2, Washington, DC, USA, 2004. IEEE Computer Society.

[140] Texas Instruments. TMSC320C55x DSP Mnemonic Instruction Set Reference Guide, 2002.

[141] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[142] Roy F. Touzeau. A fortran compiler for the FPS-164 scientific computer. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 48–57, 1984.

[143] J.W. van de Waerdt, S. Vassiliadis, S. Das, S. Mirolo, C. Yen, B. Zhong, C. Basto, J.P. van Itegem, D. Amirtharaj, K. Kalra, et al. The TM3270 media-processor. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, page 342. IEEE Computer Society, 2005.

[144] George Varghese, Sanjeev Jahagirdar, Chao Tong, Satish Damaraju Smits, Ken, Scott Siers, Ves Naydenov, Tanveer Khondker, Sanjib Sarkar, and Puneet Singh. Penryn: 45-nm next generation intel core 2 processor. In *Solid-State Circuits Conference, 2007. ASSCC '07. IEEE Asian*, pages 14 –17, Nov. 2007.

[145] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269, 1967.

[146] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. *SIGARCH Comput. Archit. News*, 20(2):256–266, 1992.

[147] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[148] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[149] Colin Whitby-Strevens. The transputer. In *ISCA '85: Proceedings of the 12th annual international symposium on Computer architecture*, pages 292–300, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[150] Mark Woh, Sangwon Seo, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. Anysp: anytime anywhere anyway signal processing. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 128–139, New York, NY, USA, 2009. ACM.

[151] Andrew Wolfe and John P. Shen. A variable instruction stream extension to the VLIW architecture. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 2–14, New York, NY, USA, 1991. ACM.

[152] Kenneth C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, 1996.

[153] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Moshenin, M. Singh, and B. Baas. An asynchronous array of simple processors for DSP applications. In *Solid-State Circuits, 2006 IEEE International Conference Digest of Technical Papers*, pages 1696–1705, 2006.

[154] Javier Zalamea, Josep Llosa, Eduard Ayguadé, and Mateo Valero. Two-level hierarchical register file organization for VLIW processors. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, pages 137–146. ACM Press, 2000.

[155] Hongtao Zhong, Kevin Fan, Scott Mahlke, and Michael Schlansker. A distributed control path architecture for VLIW processors. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 197–206, Washington, DC, USA, 2005. IEEE Computer Society.

[156] Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 25–36, Washington, DC, USA, 2007. IEEE Computer Society.

[157] Ahmad Zmily and Christos Kozyrakis. Energy-efficient and high-performance instruction fetch using a block-aware ISA. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pages 36–41, New York, NY, USA, 2005. ACM.