Vector-Thread Architecture And Implementation

by

Ronny Meir Krashinsky

B.S. Electrical Engineering and Computer Science
University of California at Berkeley, 1999

S.M. Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2001

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2007

© Massachusetts Institute of Technology 2007. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 25, 2007

Certified by . . . .
Krste Asanović
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Vector-Thread Architecture And Implementation

by

Ronny Meir Krashinsky

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2007, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

This thesis proposes vector-thread architectures as a performance-efficient solution for all-purpose computing. The VT architectural paradigm unifies the vector and multithreaded compute models. VT provides the programmer with a control processor and a vector of virtual processors. The control processor can use vector-fetch commands to broadcast instructions to all the VPs or each VP can use thread-fetches to direct its own control flow. A seamless intermixing of the vector and threaded control mechanisms allows a VT architecture to flexibly and compactly encode application parallelism and locality. VT architectures can efficiently exploit a wide variety of loop-level parallelism, including non-vectorizable loops with cross-iteration dependencies or internal control flow.

The Scale VT architecture is an instantiation of the vector-thread paradigm designed for low-power and high-performance embedded systems. Scale includes a scalar RISC control processor and a four-lane vector-thread unit that can execute 16 operations per cycle and supports up to 128 simultaneously active virtual processor threads. Scale provides unit-stride and strided-segment vector loads and stores, and it implements cache refill/access decoupling. The Scale memory system includes a four-port, non-blocking, 32-way set-associative, 32 KB cache. A prototype Scale VT processor was implemented in 180 nm technology using an ASIC-style design flow. The chip has 7.1 million transistors and a core area of 16.6 mm^2, and it runs at 260 MHz while consuming 0.4–1.1 W.

This thesis evaluates Scale using a diverse selection of embedded benchmarks, including example kernels for image processing, audio processing, text and data processing, cryptography, network processing, and wireless communication. Larger applications also include a JPEG image encoder and an IEEE 802.11a wireless transmitter. Scale achieves high performance on a range of different types of codes, generally executing 3–11 compute operations per cycle. Unlike other architectures which improve performance at the expense of increased energy consumption, Scale is generally even more energy efficient than a scalar RISC processor.

Thesis Supervisor: Krste Asanović
Title: Associate Professor

Acknowledgments

As I graduate from 24^{th} grade and look back on my education, I am grateful to so many teachers, friends, and family members that have helped me get to this point. I feel very fortunate to have had the opportunity to complete a Ph.D.

First and foremost, thank you Krste for many years of teaching and wisdom. It has been a true privilege to work with you, and you have shaped my knowledge and understanding of computer architecture and research. Thank you for being intensely interested in the details, and for, so often, guiding me to the answer when I couldn't quite formulate the question. Scale would not have come to be without your dedication, encouragement, and steadfast conviction. To: "are you done yet?", *yes*.

Chris, in 25 words or less, your eye for elegant ideas and design has improved the work we've done together. Our conversations and debates have been invaluable. Thank you.

My thesis is primarily about the Scale project, but Scale has been a team effort. I acknowledge specific contributions to the results in this thesis at the end of the introduction in Section 1.6. I am very grateful to everyone who has contributed to the project and helped to make this thesis possible.

Many brilliant people at MIT have enhanced my graduate school experience and taught me a great deal. Special thanks to all the long-term members of Krste's group, including, Seongmoo Heo, Mike Zhang, Jessica Tseng, Albert Ma, Mark Hampton, Emmett Witchel, Chris Batten, Ken Barr, Heidi Pan, Steve Gerding, Jared Casper, Jae Lee, and Rose Liu. Also, thanks to many others who were willing to impart their knowledge and argue with me, including, Matt Frank, Jon Babb, Dan Rosendband, Michael Taylor, Sam Larsen, Mark Stephenson, Mike Gordon, Bill Thies, David Wentzlaff, Rodric Rabbah, and Ian Bratt. Thanks to my thesis committee members, Arvind and Anant Agarwal. Finally, thank you to our helpful administrative staff, including Shireen Yadollahpour and Mary McDavitt.

To my friends and family: thank you so much for your love and support. This has been a long journey and I would not have made it this far without you!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer architecture is entering a new era of *all-purpose computing*, driven by the convergence of general-purpose and embedded processing.

General-purpose computers have historically been designed to support large and extensible software systems. Over the past 60 years, these machines have steadily grown smaller, faster, and cheaper. They have migrated from special rooms where they were operated by a dedicated staff, to portable laptops and handhelds. A single-chip processor today has more computing power than the fastest supercomputer systems of only 15–20 years ago. Throughout this evolution, the defining feature of these machines has been their general-purpose programmability. Computer architecture research and development has primarily targeted the performance of general-purpose computers, but power limitations are rapidly shifting the focus to efficiency.

Embedded computing devices have historically performed a single fixed function throughout the lifetime of the product. With the advent of application-specific integrated circuits (ASICs), customized chips that are dedicated to a single task, the past 25 years has seen special-purpose hardware proliferate and grow increasingly powerful. Computing is now embedded in myriad devices such as cell-phones, televisions, cars, robots, and telecommunications equipment. Embedded computing has always focused on minimizing the consumption of system resources, but chip development costs and sophisticated applications are placing a new premium on programmability.

Today's applications demand extremely high-throughput information processing in a power-constrained environment. Yet even while maximizing performance and efficiency, devices must support a wide range of different types of computation on a unified programmable substrate. This thesis presents a new all-purpose architecture that provides high efficiency across a wide range of computing applications. The vector-thread architecture (VT) merges vector and threaded execution with a vector of virtual processors that can each follow their own thread of control. VT can parallelize more codes than a traditional vector architecture, and it can amortize overheads more efficiently than a traditional multithreaded architecture.

Thesis Contributions

This thesis presents a top-to-bottom exploration of vector-thread architectures:

- **The vector-thread architectural paradigm** is a performance-efficient solution for all-purpose computing. VT unifies the vector and multithreaded execution models, allowing software to compactly encode fine-grained loop-level parallelism and locality. VT implementations simultaneously exploit data-level, thread-level, and instruction-level parallelism to provide high performance while using locality to amortize overheads.

- **The Scale VT architecture** is an instantiation of the vector-thread paradigm targeted at low-power and high-performance embedded systems. Scale's software-interface and microarchitecture pin down all the details required to implement VT, and the design incorporates many novel mechanisms to integrate vector processing with fine-grained multithreading. This thesis provides a detailed evaluation based on a diverse selection of embedded benchmarks, demonstrating Scale's ability to expose and exploit parallelism and locality across a wide range of real-world tasks.

- **The Scale VT processor** is a chip prototype fabricated in 180 nm technology. This implementation proves the feasibility of VT, and demonstrates the performance, power, and area efficiency of the Scale design.

1.1 The Demand for Flexible Performance-Efficient Computing

High-Throughput Information Processing. The miniaturization of increasingly powerful computers has driven a technological revolution based on the digitization of information. In turn, the digital world creates an ever growing demand for sophisticated high-throughput information processing. Consider just a few example devices that require flexible, performance-efficient computing:

- Over the past 10–15 years cell phones have evolved from simple voice communication devices to full-fledged multimedia computer systems. Modern "phones" provide voice, audio, image, and video processing, 3D graphics, and Internet access. They have high-resolution color displays and they support multiple high-capacity data storage mediums. These devices run operating systems with virtualization and they provide a general-purpose application programming environment.

- At one time the electronics in a television set simply projected an analog signal onto a CRT display. Today, televisions execute billions of compute operations per second to transform a highly compressed digital signal at 30 frames per second for displays with millions of pixels. Meanwhile, digital video recorders transcode multiple channels simultaneously for storage on hard disk drives. These set-top boxes connect to the Internet to retrieve programming information and run commodity operating systems to manage their computing resources.

- Modern automobiles embed many processors to replace mechanical components and provide new and improved capabilities. Computer-controlled engines improve efficiency and reduce emissions, while computer-controlled steering and braking systems improve safety. Many cars today provide cellular telephone service, speech recognition systems, and digital audio and video entertainment systems. Global positioning (GPS) receivers track location, and on-board computers provide maps and route planning. Recent advances include early-warning crash detection systems and robotic navigation systems built on extremely sophisticated processing of data from sonar sensors, radar sensors, laser scanners, and video cameras [SLB06].

- The Internet backbone is built on packet routers with extremely high data rates. The network processors in the router nodes examine incoming packets to determine destination ports and, in some cases, priorities. Higher-level processing in the routers handles packets which require special treatment. Routers may also scan packet data to detect security threats or perform other content-dependent processing. Programmability allows network processors to adapt to changing requirements, but the high data rates present a challenge for software processing.

- While increasingly powerful computing is being embedded in every-day consumer devices, conventional desktop and laptop computers are evolving to provide many of the same capabilities. Voice over Internet Protocol (VoIP) technology allows a computer to function as a telephone, and a proliferation of video over IP allows computers to function as TVs.

The Power Problem. All high-performance computing devices face stringent power constraints, regardless of whether they are portable or plugged into the power grid. This drives the need for efficient computing architectures which exploit parallelism and amortize overheads.

For battery-powered mobile platforms, energy consumption determines the lifetime between charges. Thus, the capabilities which a device is able to provide directly depend on the power usage, and the size and weight of devices depend on the associated charge storage requirements. For infrastructure processors that are connected to the power grid, energy consumption is a growing concern due to high electricity costs, especially for large data processing centers [FWB07]. Furthermore, power draw often limits a chip's maximum operating frequency. If a chip runs too fast, its cooling systems will not be able to keep up and dissipate the generated heat.

1.2 Application-Specific Devices and Heterogeneous Systems

With the growing demand for high-throughput computing in a power-constrained environment, application-specific custom solutions are an obvious choice. However, application complexity and economic forces in the chip design industry are driving current designs towards more flexible programmable processors. For very demanding applications, designers must construct heterogeneous systems with different types of computing cores.

ASICs. High-performance embedded systems have traditionally used ASICs to satisfy their demanding computing requirements with efficient custom solutions. However, increasing mask and development costs for advanced fabrication technologies have made application-specific design infeasible for all but the highest volume markets [Cha07]. With costs for next-generation chip designs reaching $30 million, the number of new ASIC design starts per year has fallen by approximately two thirds over the past 10 years [LaP07]. Furthermore, even for high-volume applications, hardwired circuits are often precluded by the growing need for flexibility and programmability.

Domain-Specific Processors. ASICs are being replaced by domain-specific processors—also known as application specific standard products (ASSPs)—which target wider markets [Ful05]. Example ASSPs include digital signal processors (DSPs), video and audio processors, and network processors. The higher volumes for these multi-purpose chips justifies the large design effort required to build highly tuned implementations, making it even harder for application-specific designs to compete. Plus, the availability of existing domain-specific processors is yet another reason to avoid the long development times required for custom chips. Many modern systems, however, have computing needs that span multiple domains. Domain-specific processors are, by definition, less efficient at tasks outside their domain. This requires systems to include multiple types of domain-specific processors, which increases complexity.

FPGAs. Field-programmable gate arrays provide an alternative computing solution between fixed-function hardware and programmable software implementations. When an application does not map well to existing programmable parts, FPGAs can provide high levels of parallelism and performance at low cost. FPGAs are programmed using hardware design languages and the code is mapped using CAD tools, which is considerably more difficult than software programming. FPGAs and general-purpose processors can be integrated in various ways to improve performance and programmability.

They can be placed together on the same board [THGM05] or the same chip [HW97, Xil05], or a programmable gate array can be integrated with a processor pipeline as a configurable functional unit [YMHB00, Str, Bar04].

Heterogeneous Systems. Full embedded systems with complex workloads and a variety of processing tasks often employ a heterogeneous mix of processing units, including ASICs, FPGAs, ASSPs, and programmable general-purpose processors. The resulting devices contain various processing cores with different instruction sets and with different parallel execution and synchronization models. Memory is typically partitioned, and communication between units goes through various point-to-point or bus-based channels. The lack of a common programming model and the lack of shared memory make these distributed systems difficult to program. They are also inefficient when the application workload causes load imbalance across the heterogeneous cores.

1.3 The Emergence of Processor Arrays

Processor arrays are emerging as a computing solution across a wide variety of embedded and general-purpose designs. Compared to application-specific solutions, programmable processor cores make a chip more flexible at executing a wider range of applications. They can also improve efficiency by allowing the same silicon area to be reused for different compute tasks depending on the needs of an application as it runs. Compared to heterogeneous designs, homogeneous processor arrays simplify programming and enable more portable and scalable software. Shared memory and a unified synchronization model can further simplify programming complexity compared to designs with ad-hoc communication mechanisms.

The move towards processor arrays is also driven by physical trends in chip design. With device counts in modern chips surpassing 1 billion transistors, the logical complexity of monolithic designs has become prohibitive. The advent of gigahertz-level clock frequencies has made wire delay a significant factor, with signals taking several cycles to cross a chip [TKM$^+$02]. Increasing chip integration coupled with limited supply voltage scaling and leaky transistors has made power a primary design consideration. Re-use helps to reduce complexity, and partitioning the design lessens the impact of wire delay and reduces energy consumption.

In the general-purpose computing arena, industry has moved towards multi-core chips as the new way to extract performance from ever-increasing transistor counts. General-purpose superscalar processors have microarchitectural structures which scale poorly as clock frequencies increase [AHKB00]. Furthermore, even when clock rates remain unchanged, these processors must consume large amounts of silicon area and power to achieve modest improvements in instruction-level parallelism [ONH$^+$96]. The thread-level parallelism exploited by multi-core superscalars has emerged as a more feasible way to scale performance [SACD04]. Current designs include AMD's quad-core Opteron [DSC$^+$07], IBM's dual-core POWER6 [FMJ$^+$07], and Intel's dual-core Xeon processor [RTM$^+$07]. Server applications generally have higher levels of thread parallelism, and Sun's Niagara 2 chip [McG06] uses 8 cores that can each run 8 threads simultaneously.

Embedded designs are using even larger-scale processor arrays. To meet the high-throughput computing needs of games machines and other embedded applications, IBM's Cell chip includes a general-purpose processor and an array of 8 SIMD processing engines [GHF$^+$06]. Graphics processing units (GPUs) are transitioning from fixed-function chips to increasingly flexible computing substrates. For example, NVidia's GeForce 8 merges the previously hard-wired pixel and vertex shader functionality into a homogeneous array of 128 programmable processors [NVI06]. Other recent embedded processor chips include Cavium Networks' 16-core network processor [YBC$^+$06], Cisco's network processor with 188 Tensilica Xtensa cores [Eat05], Ambric's 360-processor

AM2045 [Hal06a], PicoChip's 430-processor PC101 DSP array [DPT03], and Connex Technology's video processing chips with up to 4096 processing elements [Hal06b].

Even FPGAs are evolving in the direction programmable processor arrays. Current Xilinx parts include hundreds of DSP functional units embedded in the gate array [Xil07], and MathStar's field programmable object array (FPOA) uses hundreds of arithmetic units and register files as configurable building blocks [Hal06c].

1.4 An Efficient Core for All-Purpose Computing

The growing performance and efficiency requirements of today's computing applications and the increasing development costs for modern chips drive the need for all-purpose programmable architectures. Recently, high-bandwidth information processing demands and complexity limitations have led designers to build chips structured as processor arrays. In these designs, a homogeneous programming model with uniform communication and synchronization mechanisms can ease the programming burden. From the perspective of this thesis, broad classes of applications will converge around a small number of programmable computing architectures. Future chips will be constructed as arrays of performance-efficient flexible cores.

The processor architectures widely used today will not necessarily meet the unique demands of future embedded cores. As a replacement for special-purpose hardware, programmable processors will have to attain very high levels of performance and efficiency. Given the local instruction and data caches and on-chip network connections which a core will require, a straightforward scalar RISC processor probably does not provide enough compute bandwidth to amortize these overheads. Additionally, although they are simple, scalar processors are not as efficient as possible since they do not fully exploit application parallelism and locality. Furthermore, very small cores can make the software thread parallelization challenge more difficult.

Scaling conventional general-purpose cores is also not likely to lead to performance-efficient solutions for embedded processing. Sequential architectures provide minimal support for encoding parallelism or locality, so high-performance implementations are forced to devote considerable area and power to on-chip structures that track dependencies or that provide arbitrary global communication. For example, Intel's recent Penryn processor fits two superscalar cores in about 44 mm^2 in 45 nm technology [Int07]. In comparison, the Scale VT processor described in this thesis can execute 16 instructions per cycle and support up to 128 simultaneously active threads. In 45 nm technology Scale would only be about 1 mm^2, allowing over 40 cores to fit within the same area as Penryn's two superscalar cores. Although current general-purpose architectures devote large amounts of silicon area to improve single-thread performance, compute density will become increasingly important for future processor arrays.

Ideally, a single all-purpose programmable core will efficiently exploit the different types of parallelism and locality present in high-throughput information processing applications. These applications often contain abundant structured parallelism, where dependencies can be determined statically. Architectures that expose more parallelism allow implementations to achieve high performance with efficient microarchitectural structures. Similarly, architectures that expose locality allow implementations to isolate data communication and amortize control overheads.

1.5 Thesis Overview

This thesis begins by proposing the vector-thread architectural paradigm as a solution for all-purpose computing. Chapter 2 describes how VT architectures unify the vector and multithreaded execution models to flexibly and compactly encode large amounts of structured parallelism and locality. VT is primarily targeted at efficiently exploiting all types of loop-level parallelism, and Chapter 3 compares VT to other architectures.

This thesis then presents the Scale architecture, an instantiation of the vector-thread paradigm designed for low-power and high-performance embedded systems. Chapter 4 presents the software interface for Scale and discusses alternatives and implications for VT instruction sets. VT machines use many novel mechanisms to integrate vector processing with fine-grained multithreading and to amortize the control and data overheads involved. Chapter 5 details a complete microarchitecture of a Scale implementation and discusses some of the design decisions.

The third major component of this thesis is a description of the prototype Scale VT processor implementation. Chapter 6 provides an overview of the chip design and verification flow which allowed us to implement Scale with relatively low design effort. The Scale chip has been fabricated in 180 nm technology, and the chapter presents area, clock frequency, and power results.

The final component of this thesis is a detailed evaluation of the Scale vector-thread architecture presented in Chapters 7 and 8. To evaluate the performance and flexibility of Scale, we have mapped a diverse selection of embedded benchmarks from various suites. These include example kernels for image processing, audio processing, cryptography, network processing, and wireless communication. Larger applications including a JPEG image encoder and an IEEE 802.11a wireless transmitter have also been programmed on Scale. The chapter discusses how various forms of application parallelism are mapped to the Scale VT architecture, and it presents both simulation results and hardware measurements. The chapter also analyzes the locality and efficiency properties of the Scale architecture and the benchmark mappings. Chapter 8 evaluates the performance and efficiency of Scale's memory system, with a focus on cache refill/access decoupling.

1.6 Collaboration, Previous Publications, and Funding

This thesis describes work that was performed as part of a collaborative group project. Other people have made direct contributions to the ideas and results that are included in this document. Christopher Batten played a large role in the development of the Scale architecture. He was also responsible for the Scale cache, including both the simulator and the chip, and he worked on the datapath tiler used for constructing the chip. Mark Hampton was responsible for the Scale compiler tool-chain. He also implemented the VTorture chip testing tool. Steve Gerding designed the XBS benchmark suite. Jared Casper designed a controller for the test baseboard and worked on an initial implementation of the cache design. The benchmarks in this thesis were mapped to Scale by Steve, Jared, Chris, Mark, Krste, and myself. Albert Ma designed the VCO for the clock on the Scale chip and he helped to develop the CAD tool-flow. Asif Khan implemented the DRAM controller on the test baseboard. Jaime Quinonez helped to implement the datapath tiler. Brian Pharris wrote the DRAM model used in the Scale simulator. Jeff Cohen implemented an initial version of VTorture. Ken Barr wrote the host driver which communicates with the test baseboard. Finally, Krste Asanović was integral in all aspects of the project.

Some of the content in this thesis is adapted from previous publications, including: "The Vector-Thread Architecture" from *ISCA, 2004* [KBH$^+$04a], "The Vector-Thread Architecture" from *IEEE Micro, 2004* [KBH$^+$04b], and "Cache Refill/Access Decoupling for Vector Machines" from *MICRO, 2004* [BKGA04].

This work was funded in part by DARPA PAC/C award F30602-00-2-0562, NSF CAREER award CCR-0093354, and donations from Infineon Corporation and Intel Corporation.

Chapter 2

The Vector-Thread Architectural Paradigm

In this chapter we introduce the vector-thread architectural paradigm. VT supports a seamless intermingling of vector and multithreaded computation to allow software to flexibly and compactly encode application parallelism and locality. VT implementations exploit this encoding to provide high performance while amortizing instruction and data overheads. VT builds on aspects of RISC, vector, and multiprocessor architectures, and we first describe these paradigms in order to set the stage for VT. We then introduce the vector-thread paradigm, including the programmer's model and the physical model for VT machines, as well as the execution semantics for vector-thread programs. VT is an abstract class of computer architectures, and in this chapter we also describe several possible extensions to the baseline design. Finally, we provide an overview of vector-threading software for VT architectures.

2.1 Defining Architectural Paradigms

A vast gulf separates algorithms—theoretical expressions of computation—from the transistors, wires, and electrons in computing hardware. Multiple layers of software and hardware abstractions bridge this gap and these can be categorized into a few computing and architectural paradigms. This section defines the relationship between these abstractions.

A *computing paradigm* describes the structure and organization of control flow and data movement in a computer system. Von Neumann computing is the most common paradigm in use today [vN93]. It includes a uniform global memory, and programs are constructed as an ordered sequence of operations directed by imperative control constructs. Multithreaded programming is an extension to von Neumann computing where multiple threads of control execute programs simultaneously. In the shared memory variation, all of the threads operate concurrently and communicate through a shared global memory. In the message-passing variation, the threads operate on independent memories and communicate with each other by sending explicit messages. Further afield from von Neumann computing, the dataflow paradigm has no global memory and no explicit control flow. A dataflow program consists of a set of interconnected operations which execute whenever their input operands are available. A related model is the stream computing paradigm where data implicitly flows through interconnected computational kernels.

Programming languages allow people to precisely encode algorithms at a high level of abstraction. They generally follow a particular computing paradigm. For example, FORTRAN, C, C++, and Java are all von Neumann programming languages. Multithreaded computing paradigms are often targeted by programming libraries rather than built-in language constructs. Examples include POSIX threads [IEE95] for shared memory multithreading, and MPI [For94] for multithreading with message passing.

An *instruction set architecture* (ISA) defines a low-level hardware/software interface, usually in the form of machine instructions. For example, MIPS-IV is an ISA which is specified by around 300 pages of documentation [Pri95]. Compilers map high-level programming languages to machine code for a specific ISA, or assembly programmers may write the code directly. Generally, many different machines implement a particular ISA. The MIPS R10000 [Yea96] is an example microprocessor which implements the MIPS-IV ISA.

Broader than any specific instruction set architecture, an *architectural paradigm* includes a programmer's model for a class of machines together with the expected structure of implementations of these machines. The programming model is an abstract low-level software interface that gives programmers an idea of how code executes on the machines. There is also usually an *archetype* for the paradigm, a canonical implementation which shows the general structure of the category of machines. Although specific instruction set architectures and machine implementations are often hybrids or variations of architectural paradigms, the abstract classification is still important. It serves as a useful model for programmers, languages, and compilers, and it enables a generalized discussion of the performance, efficiency, and programmability merits of different architectures.

Architectural paradigms are usually designed to efficiently support specific computing paradigms, or sometimes vice-versa. But in general, any computing paradigm or language can map to any architectural paradigm. Popular architectural paradigms for von Neumann computing include RISC (reduced instruction set computing), superscalar, VLIW (very long instruction word), subword-SIMD (single instruction multiple data), and vector. Popular architectural paradigms for shared memory multiprocessing include symmetric multiprocessors (SMPs) and simultaneous multithreading (SMT). The following sections describe several of these architectural paradigms in more detail.

2.2 The RISC Architectural Paradigm

RISC is an architectural paradigm which supports von Neumann computing and provides a simple instruction interface to enable high-speed hardware implementations. The programmer's model for a RISC architecture, diagrammed in Figure 2-1, includes a generic set of registers and a set of instructions which operate on those registers. For example, an add instruction takes two source registers as operands and writes its result to a destination register. Load and store instructions use registers to compute an address and transfer data between main memory and the register file. A program for a RISC architecture is organized as a sequence of instructions, and a RISC machine has a program counter which contains the address of the current instruction. By default, the machine implicitly increments the program counter to step through the instruction sequence in the program. Branch and jump instructions direct the program control flow by providing a new address for the program counter. Conditional branches based on the values of registers allow programs to have data-dependent control flow.

The RISC archetype (Figure 2-2) includes a register file and an ALU (arithmetic logic unit). It also has instruction and data caches which provide fast access to recently used data, backed up by a large main memory. As instructions execute, they progress through 5 stages: fetch, decode, execute, memory access, and writeback. A pipelined implementation organizes each stage as a single clock cycle, enabling a fast clock frequency and allowing the execution of multiple instructions to be overlapped. When an instruction writes a register that is read by the following instruction, the result

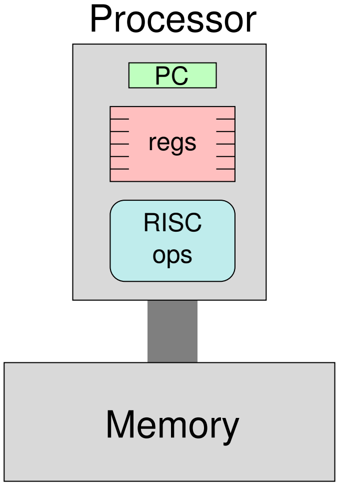


Figure 2-1: Programmer's model for a generic RISC architecture. The processor includes a set of a registers and a program counter. It executes simple instructions and connects to a uniform memory.

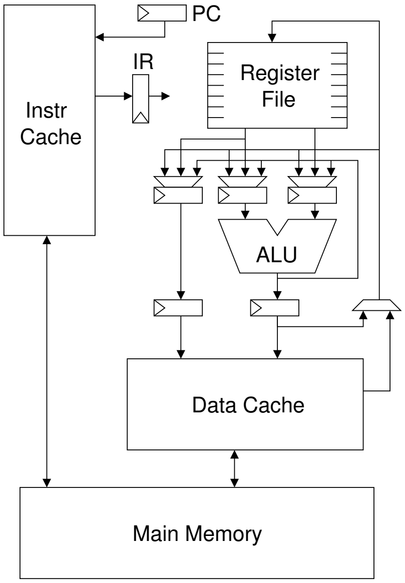


Figure 2-2: RISC archetype. The pipelined processor includes a register file and an ALU. Instruction and data caches provide fast access to memory.

may be bypassed in the pipeline before it is written to the register file. Since conditional branch instructions do not resolve until the decode or execute stage of the pipeline, a pipelined machine may predict which instruction to fetch each cycle. When mispredictions are detected, the incorrect instructions are flushed from the pipeline and the program counter is set to the correct value. The simplest scheme predicts sequential control flow, but more complex schemes can predict taken jumps and branches.

Figure 2-4 shows an example of RISC assembly code for the RGB to YCbCr color conversion kernel shown in C in Figure 2-3. This kernel converts an input image from an interleaved RGB (red-green-blue) format to partitioned luminance (Y) and chrominance components (Cb, Cr), a format used in image and video processing applications. Figure 2-5 shows a simple overview of how the code executes on a pipelined RISC machine.

```
void RGBtoYCbCr (uint8_t* in_ptr,
                 uint8_t* outy_ptr,
                 uint8_t* outcb_ptr,
                 uint8_t* outcr_ptr,
                 int n) {
  for (int i=0; i<n; i++) {
    uint8_t r = *in_ptr++;
    uint8_t g = *in_ptr++;
    uint8_t b = *in_ptr++;
    *outy_ptr++  = (((r*CONST_Y_R )+(g*CONST_Y_G )+(b*CONST_Y_B )+CONST_Y_RND) >> 16);
    *outcb_ptr++ = (((r*CONST_CB_R)+(g*CONST_CB_G)+(b*CONST_CB_B)+CONST_C_RND) >> 16);
    *outcr_ptr++ = (((r*CONST_CR_R)+(g*CONST_CR_G)+(b*CONST_CR_B)+CONST_C_RND) >> 16);
  }
}
```

Figure 2-3: RGB-to-YCC color conversion kernel in C. The code uses 8-bit fixed-point datatypes.

```
RGBtoYCbCr:
loop:

  lbu  r1, 0(rin)          # load R
  lbu  r2, 1(rin)          # load G
  lbu  r3, 2(rin)          # load B

  mul  r4, r1, CONST_Y_R   # R * Y_R
  mul  r5, r2, CONST_Y_G   # G * Y_G
  add  r6, r4, r5          # accumulate Y
  mul  r4, r3, CONST_Y_B   # B * Y_B
  add  r6, r6, r4          # accumulate Y
  add  r6, r6, CONST_Y_RND # round Y
  srl  r6, r6, 16          # scale Y
  sb   r6, 0(routy)        # store Y

  mul  r4, r1, CONST_CB_R  # R * CB_R
  mul  r5, r2, CONST_CB_G  # G * CB_G
  add  r6, r4, r5          # accumulate CB
  mul  r4, r3, CONST_CB_B  # B * CB_B
  add  r6, r6, r4          # accumulate CB
  add  r6, r6, CONST_C_RND # round CB
  srl  r6, r6, 16          # scale CB
  sb   r6, 0(routcb)       # store CB

  mul  r4, r1, CONST_CR_R  # R * CR_R
  mul  r5, r2, CONST_CR_G  # G * CR_G
  add  r6, r4, r5          # accumulate CR
  mul  r4, r3, CONST_CR_B  # B * CR_B
  add  r6, r6, r4          # accumulate CR
  add  r6, r6, CONST_C_RND # round CR
  srl  r6, r6, 16          # scale CR
  sb   r6, 0(routcr)       # store CR

  add  rin, rin, 3         # incr rin
  add  routy, routy, 1     # incr routy
  add  routcb, routcb, 1   # incr routcb
  add  routcr, routcr, 1   # incr routcr
  sub  n, n, 1             # decr n

  bnez n, loop             # loop
  jr   ra                  # func. return
```

Figure 2-4: RGB-to-YCC in generic RISC assembly code. For clarity, some registers are referred to using names instead of numbers (e.g. `rin`, `routy`, `n`).

```
               F   D   X   M   W
cycle 1:     lbu
cycle 2:     lbu lbu
cycle 3:     lbu lbu lbu
cycle 4:     mul lbu lbu lbu
cycle 5:     mul mul lbu lbu lbu
cycle 6:     add mul mul lbu lbu
cycle 7:     mul add mul mul lbu
cycle 8:     add mul add mul mul
cycle 9:     add add mul add mul
[ ... ]
```

Figure 2-5: RGB-to-YCC pipelined execution on a RISC machine. Each horizontal line corresponds to a single clock cycle, and the columns show which instruction is in each stage of the pipeline.

2.3 The Symmetric Multiprocessor Architectural Paradigm

The SMP architectural paradigm uses thread parallelism to increase processing performance compared to a single-threaded machine. SMP architectures provide a set of processor nodes with symmetric access to a global shared memory. The individual nodes can use any architecture, e.g. RISC in the example shown in Figure 2-6. SMPs support a multithreaded computing paradigm. The programmer's model consists of a set of independent threads which execute in the same memory space, and threads can use regular loads and stores to share data. Synchronization between threads is an important concern, and an SMP architecture usually provides atomic read-modify-write memory operations to enable software-based synchronization primitives such as locks and semaphores.

In an SMP machine, such as the archetype shown in Figure 2-7, processors have independent caches, and memory is often distributed across the processing nodes. To support the shared memory model where stores by one processor should become visible to loads of the same location by other processors, SMPs must implement cache-coherence algorithms. A baseline algorithm, MSI, tracks the state of cache lines in each node as modified (M), shared (S), or invalid (I). Different nodes can

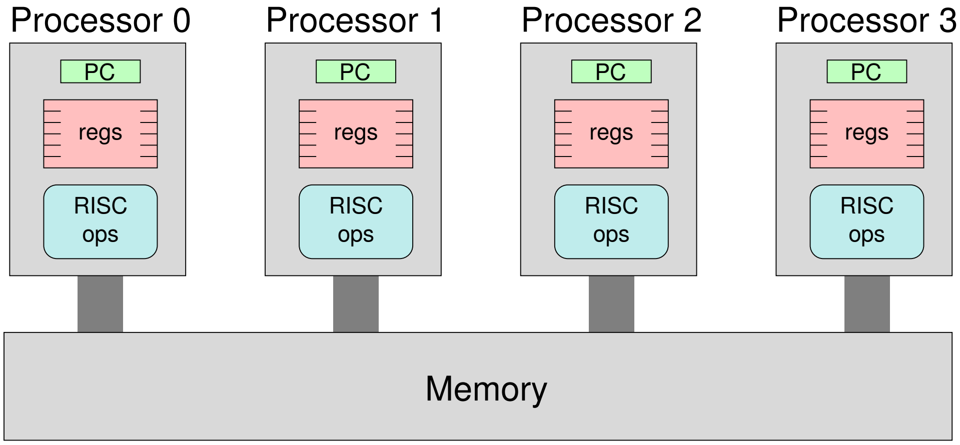


Figure 2-6: Programmer's model for a generic SMP architecture. In this example a set of 4 RISC processors share a common memory.

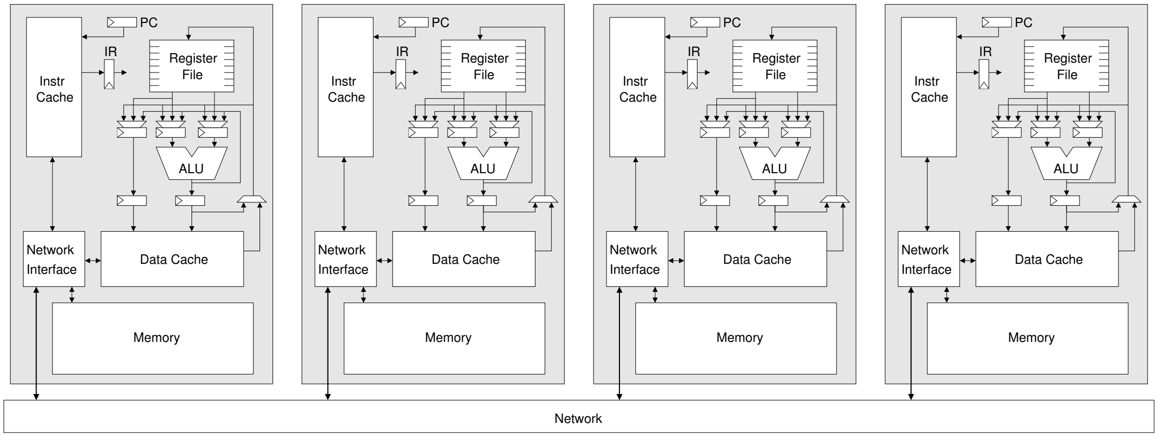


Figure 2-7: SMP archetype. In this example, four processing nodes each include a RISC processor and a slice of the system memory. A network connects the nodes together and serves as a conduit for cache coherence messages.

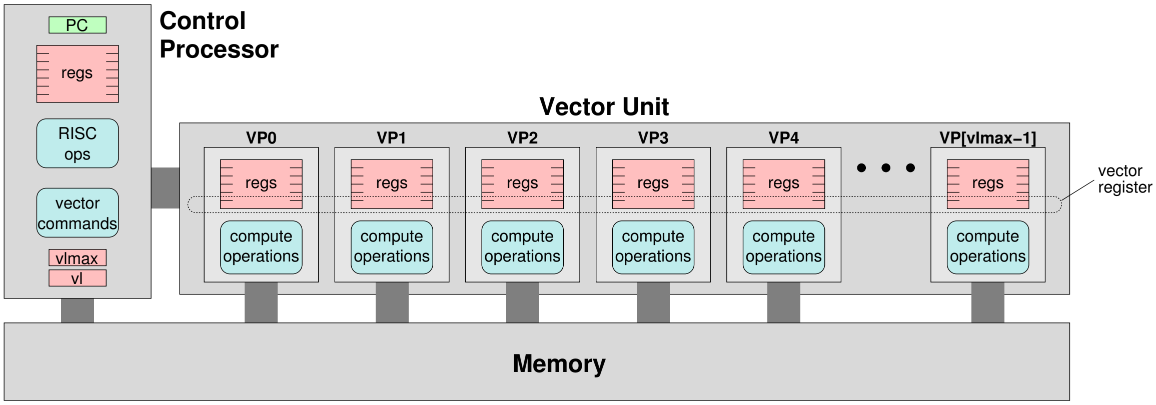


Figure 2-8: Programmer's model for a generic vector architecture (virtual processor view). A vector of virtual processors each contain registers and have the ability to execute basic operations. A RISC processor with support for vector commands controls the VPs en masse.

read the same memory locations, and they acquire the associated lines in the shared state. However, a node can only write to a memory location if it has the cache line in the modified state. Only one node at a time is allowed to have a line in this state, so a node may need to invalidate the line in other nodes' caches before it can complete a write. This can be accomplished with a snoopy bus protocol in which each node observes the memory accesses of all other nodes and responds appropriately. Or, a directory may be used to track the state of cache lines in the system, with nodes sending requests to each other to change the states of lines.

2.4 The Vector Architectural Paradigm

The vector architectural paradigm exploits data parallelism to enhance performance and efficiency. The programmer's model for a vector architecture (Figure 2-8) extends a traditional RISC control processor with vector registers and vector instructions. Each vector register contains many elements, and vector instructions perform operations in parallel across these elements. An easy way to think about the resources in a vector machine is as a virtual processor vector [ZB91], where a virtual processor contains an element-wise subset of the vector registers. For example, a vector-vector-add:

```
vvadd vr3, vr1, vr2
```

will compute an addition in each virtual processor:

VP0.vr3 = VP0.vr1 + VP0.vr2
VP1.vr3 = VP1.vr1 + VP1.vr2
...
VPN.vr3 = VPN.vr1 + VPN.vr2

Vector loads and stores transfer data between memory and the vector registers, with each vector memory instruction loading or storing one element per VP. With unit-stride and strided vector memory instructions, the control processor provides the address information, for example:

```
vlwst vr3, rbase, rstride # rbase and rstride are CP registers
```

VP0.vr3 = mem[rbase + 0×rstride]
VP1.vr3 = mem[rbase + 1×rstride]
...
VPN.vr3 = mem[rbase + N×rstride]

```
RGBtoYCbCr:
  mult rinstripstr, vlmax, 3
  li rstride, 24

stripmineloop:

  setvl vlen, n # (vl,vlen) = MIN(n,vlmax)

  vlbust vr1, rin, rstride   # load R
  add    r1,  rin, 1
  vlbust vr2, r1,  rstride   # load G
  add    r1,  rin, 2
  vlbust vr3, r1,  rstride   # load B

  vsmul vr4, vr1, CONST_Y_R  # R * Y_R
  vsmul vr5, vr2, CONST_Y_G  # G * Y_G
  vvadd vr6, vr4, vr5        # accum. Y
  vsmul vr4, vr3, CONST_Y_B  # B * Y_B
  vvadd vr6, vr6, vr4        # accum. Y
  vsadd vr6, vr6, CONST_Y_RND# round Y
  vssrl vr6, vr6, 16         # scale Y
  vsb   vr6, routy           # store Y

  vsmul vr4, vr1, CONST_CB_R # R * CB_R
  vsmul vr5, vr2, CONST_CB_G # G * CB_G
  vvadd vr6, vr4, vr5        # accum. CB
  vsmul vr4, vr3, CONST_CB_B # B * CB_B
  vvadd vr6, vr6, vr4        # accum. CB
  vsadd vr6, vr6, CONST_C_RND# round CB
  vssrl vr6, vr6, 16         # scale CB
  vsb   vr6, routcb          # store CB
```

```
vsmul vr4, vr1, CONST_CR_R # R * CR_R
vsmul vr5, vr2, CONST_CR_G # G * CR_G
vvadd vr6, vr4, vr5        # accum. CR
vsmul vr4, vr3, CONST_CR_B # B * CR_B
vvadd vr6, vr6, vr4        # accum. CR
vsadd vr6, vr6, CONST_C_RND# round CR
vssrl vr6, vr6, 16         # scale CR
vsb   vr6, routcr          # store CR

add   rin, rin, rinstripstr# incr rin
add   routy, routy, vlen   # incr routy
add   routcb, routcb, vlen # incr routcb
add   routcr, routcr, vlen # incr routcr
sub   n, n, vlen           # decr n

bnez  n, stripmineloop     # stripmine
jr    ra                   # func. return
```

Figure 2-9: RGB-to-YCC in generic vector assembly code.

With indexed vector memory instructions, the addresses are taken from vector registers, for example:

```
vlw vr3, vr1
```

VP0.vr3 = mem[VP0.vr1]
VP1.vr3 = mem[VP1.vr1]
...
VPN.vr3 = mem[VPN.vr1]

The maximum vector length (vlmax) is the number of virtual processors (i.e. vector register elements) provided by the vector machine. The active vector length (vl) may be changed dynamically, typically by writing a control register. A vectorizable loop is executed using stripmining, where strips of vlmax iterations of the loop are executed at a time using vector instructions. Figure 2-9 shows the RGB-to-YCC example coded for a generic vector architecture.

The archetype for a vector machine (Figure 2-10) includes a vector unit with an array of processing lanes. These lanes hold the vector registers, and the VPs (vector elements) are striped across the lanes. The control processor executes the main instruction stream and queues vector instructions in a vector command queue. The vector unit processes a vector command by issuing element groups across all the lanes in parallel each cycle. For example, with 4 lanes and a vector length of 16, a vector-vector-add will issue over 4 cycles, first for VP0, VP1, VP2, and VP3; then for VP4, VP5, VP6, and VP7; etc. This is accomplished simply by incrementing a base register file index each cycle to advance to the next VP's registers while keeping the ALU control signals fixed.

A vector unit may connect either to a data cache or directly to main memory. The vector machine includes vector load and store units (VLU, VSU) which processes unit-stride and strided vector loads and stores to generate requests to the memory system. Unit-stride instructions allow a

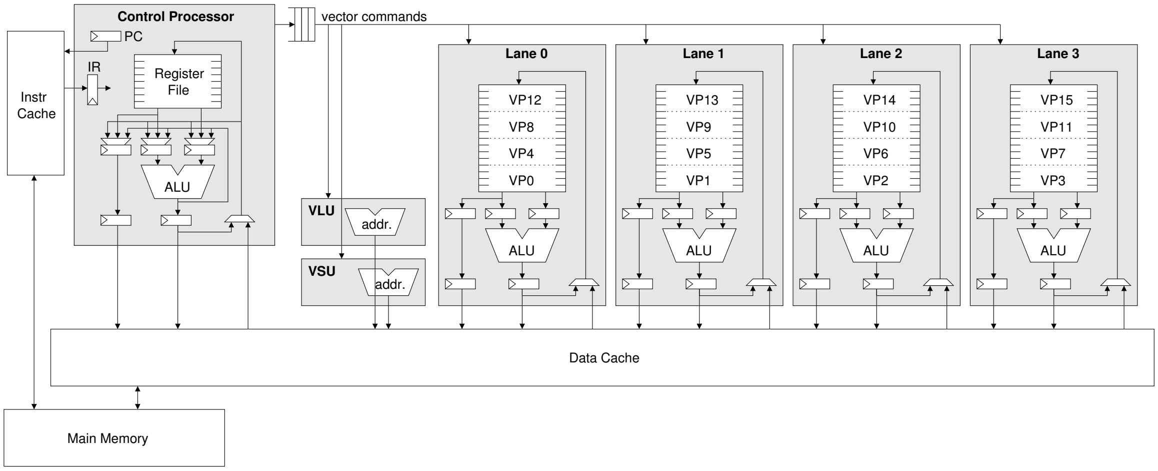


Figure 2-10: Vector archetype. The vector unit contains a parallel array of lanes with ALUs and register files. The virtual processors are striped across the lane array.

single memory access (with one address) to fetch multiple elements together. To handle indexed accesses, each lane has an address port into the memory system, allowing several independent requests to be made each cycle. Strided accesses may be implemented using the lane ports for increased throughput.

Figure 2-11 shows how the RGB-to-YCC example executes on the vector archetype. The speedup compared to a RISC machine is approximately a factor of 4, i.e. the number of lanes which execute operations in parallel. In addition to the lane parallelism, the loop bookkeeping code is factored out and executed only once per stripmine iteration by the control processor. This includes incrementing the input and output pointers, tracking the iteration count, and executing the loop branch instruction. The control processor can execute these in the background while the lanes process vector instructions, further improving the speedup compared to a RISC machine.

Although the baseline vector archetype processes only one vector instruction at a time, most vector machines provide higher throughput. Figure 2-12 shows the vector archetype extended to include two arithmetic units plus independent load and store ports. Chaining in a multi-unit vector machine allows the execution of different vector instructions to overlap. At full throughput, in each cycle a lane can receive incoming load data, perform one or more arithmetic ops, and send out store data. Figure 2-13 shows an example of chaining on a multi-unit vector machine.

2.5 The Vector-Thread Architectural Paradigm

VT Programmer's Model

The vector-thread architectural paradigm is a hybrid combining the vector and multiprocessor models. A conventional control processor interacts with a vector of virtual processors, as shown in Figure 2-14. As in a traditional vector architecture, vector commands allow the control processor to direct the VPs en masse. However, VPs also have the ability to direct their own control flow, as in a multiprocessor architecture. As a result, VT can seamlessly intermingle data-level and thread-level parallelism at a very fine granularity.

Under the VT programmer's model, there are two interacting instruction sets, one for the control processor and one for the virtual processors. A VP contains a set of registers and can execute basic

```
  CP       Lane 0    Lane 1    Lane 2    Lane 3

vlbust      0: lb     1: lb     2: lb     3: lb
add         4: lb     5: lb     6: lb     7: lb
            8: lb     9: lb    10: lb    11: lb
           12: lb    13: lb    14: lb    15: lb
vlbust      0: lb     1: lb     2: lb     3: lb
add         4: lb     5: lb     6: lb     7: lb
            8: lb     9: lb    10: lb    11: lb
           12: lb    13: lb    14: lb    15: lb
vlbust      0: lb     1: lb     2: lb     3: lb
            4: lb     5: lb     6: lb     7: lb
            8: lb     9: lb    10: lb    11: lb
           12: lb    13: lb    14: lb    15: lb
vsmul       0: *      1: *      2: *      3: *
            4: *      5: *      6: *      7: *
            8: *      9: *     10: *     11: *
           12: *     13: *     14: *     15: *
vsmul       0: *      1: *      2: *      3: *
            4: *      5: *      6: *      7: *
            8: *      9: *     10: *     11: *
           12: *     13: *     14: *     15: *
vvadd       0: +      1: +      2: +      3: +
            4: +      5: +      6: +      7: +
            8: +      9: +     10: +     11: +
           12: +     13: +     14: +     15: +
vsmul       0: *      1: *      2: *      3: *
            4: *      5: *      6: *      7: *
            8: *      9: *     10: *     11: *
           12: *     13: *     14: *     15: *
vvadd       0: +      1: +      2: +      3: +
            4: +      5: +      6: +      7: +
            8: +      9: +     10: +     11: +
           12: +     13: +     14: +     15: +
vsadd       0: +      1: +      2: +      3: +
            4: +      5: +      6: +      7: +
            8: +      9: +     10: +     11: +
           12: +     13: +     14: +     15: +
vssrl       0: >>     1: >>     2: >>     3: >>
            4: >>     5: >>     6: >>     7: >>
            8: >>     9: >>    10: >>    11: >>
           12: >>    13: >>    14: >>    15: >>
vsbst       0: sb     1: sb     2: sb     3: sb
            4: sb     5: sb     6: sb     7: sb
            8: sb     9: sb    10: sb    11: sb
           12: sb    13: sb    14: sb    15: sb
[ ... ]
```

Figure 2-11: RGB-to-YCC vector execution. Virtual processor numbers for the issuing element operations are indicated before the colon. This example assumes minimal decoupling between the control processor and the vector unit. Typically, a vector command queue would allow the control processor to run further ahead.

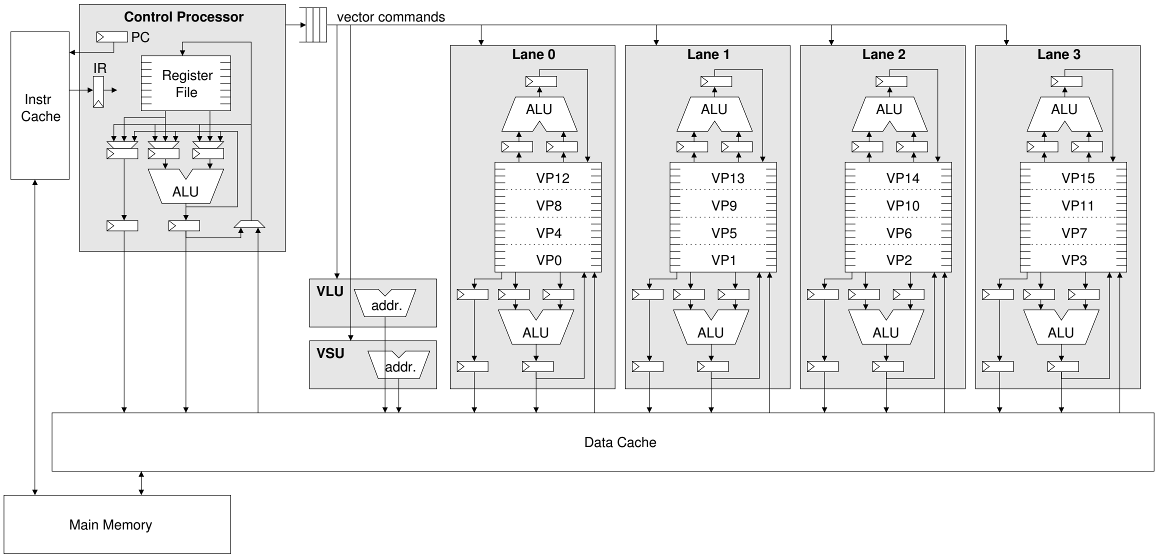


Figure 2-12: Vector archetype with two arithmetic units plus independent load and store ports.

```
  CP                   Lane 0                      Lane 1        [ ... ]

vlbust      0: lb                       1: lb
vsmul       4: lb  0: *                 5: lb  1: *
vvadd       8: lb  4: *  0: +           9: lb  5: *  1: +
vsbst      12: lb  8: *  4: +  0: sb   13: lb  9: *  5: +  1: sb
vlbust      0: lb 12: *  8: +  4: sb    1: lb 13: *  9: +  5: sb
vsmul       4: lb  0: * 12: +  8: sb    5: lb  1: * 13: +  9: sb
vvadd       8: lb  4: *  0: + 12: sb    9: lb  5: *  1: + 13: sb
vsbst      12: lb  8: *  4: +  0: sb   13: lb  9: *  5: +  1: sb
vlbust      0: lb 12: *  8: +  4: sb    1: lb 13: *  9: +  5: sb
vsmul       4: lb  0: * 12: +  8: sb    5: lb  1: * 13: +  9: sb
vvadd       8: lb  4: *  0: + 12: sb    9: lb  5: *  1: + 13: sb
vsbst      12: lb  8: *  4: +  0: sb   13: lb  9: *  5: +  1: sb
                  12: *  8: +  4: sb          13: *  9: +  5: sb
                        12: +  8: sb                13: +  9: sb
                              12: sb                      13: sb
```

Figure 2-13: Vector execution with chaining. The example assumes that every cycle each lane can receive load data, perform two arithmetic operations, and send store data. Virtual processor numbers for the issuing element operations are indicated before the colon. Only two lanes are shown.

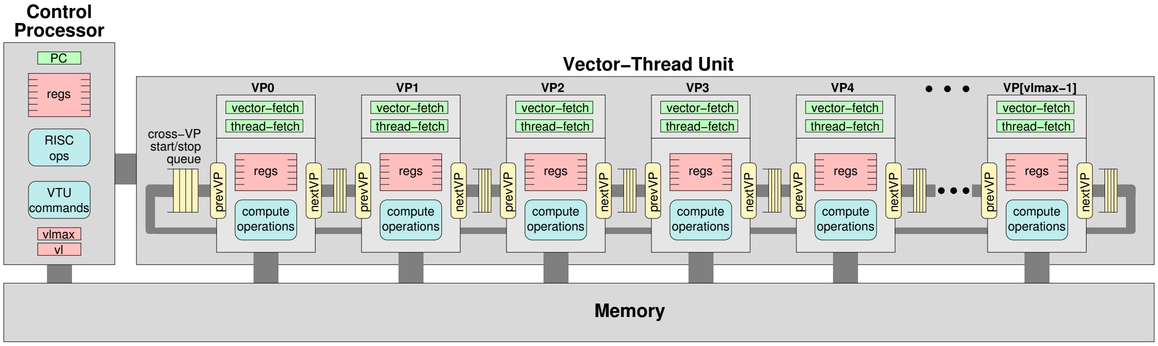


Figure 2-14: Programmer's model for a generic vector-thread architecture. As in a vector architecture, a control processor interacts with a virtual processor vector. The VPs execute instructions that are grouped into atomic instruction blocks (AIBs). AIBs can be sent to the VPs from a control processor vector-fetch command, or a VP itself can issue a thread-fetch to request an AIB. The VPs are interconnected in a unidirectional ring through prevVP and nextVP ports.

RISC instructions like loads, stores, and ALU operations. These VP instructions are grouped into atomic instruction blocks (AIBs). There is no automatic program counter or implicit instruction fetch mechanism for VPs. Instead, all instruction blocks must be explicitly requested by either the control processor or the VP itself.

When executing purely vectorizable code, the virtual processors operate as slaves to the control processor. Instead of using individual vector instructions like vector-vector-add, the control processor uses vector-fetch commands to send AIBs to the virtual processor vector. The control processor also uses traditional vector-load and vector-store commands to transfer data between memory and the VP registers. Figure 2-15 shows a diagram of vector code mapped to the VT architecture. As in a vector machine the virtual processors all execute the same element operations, but each operates on its own independent set of registers.

Multithreading in the VT architecture is enabled by giving the virtual processors the ability to direct their own control flow. A VP issues a *thread-fetch* to request an AIB to execute after it completes its active AIB, as shown in Figure 2-16. VP fetch instructions may be unconditional, or they may be predicated to provide conditional branching. A VP thread persists as long as each AIB contains an executed fetch instruction, but halts when an AIB does not issue a thread-fetch to bring in another AIB.

The VT programmer's model also connects VPs in a unidirectional ring topology and allows a sending instruction on VP$_n$ to transfer data directly to a receiving instruction on VP$_{n+1}$. These *cross-VP data transfers* are dynamically scheduled and resolve when the data becomes available. As shown in Figure 2-17, a single vector-fetch command can introduce a chain of prevVP receives and nextVP sends that spans the virtual processor vector. The control processor can push an initial value into the *cross-VP start/stop queue* (shown in Figure 2-14) before executing the vector-fetch command. After the chain executes, the final cross-VP data value from the last VP wraps around and is written into this same queue. It can then be popped by the control processor or consumed by a subsequent prevVP receive on VP0 during stripmined loop execution.

VT Archetype

The VT archetype contains a conventional scalar unit for the control processor together with a *vector-thread unit* (VTU) that executes the VP code. To exploit the parallelism exposed by the

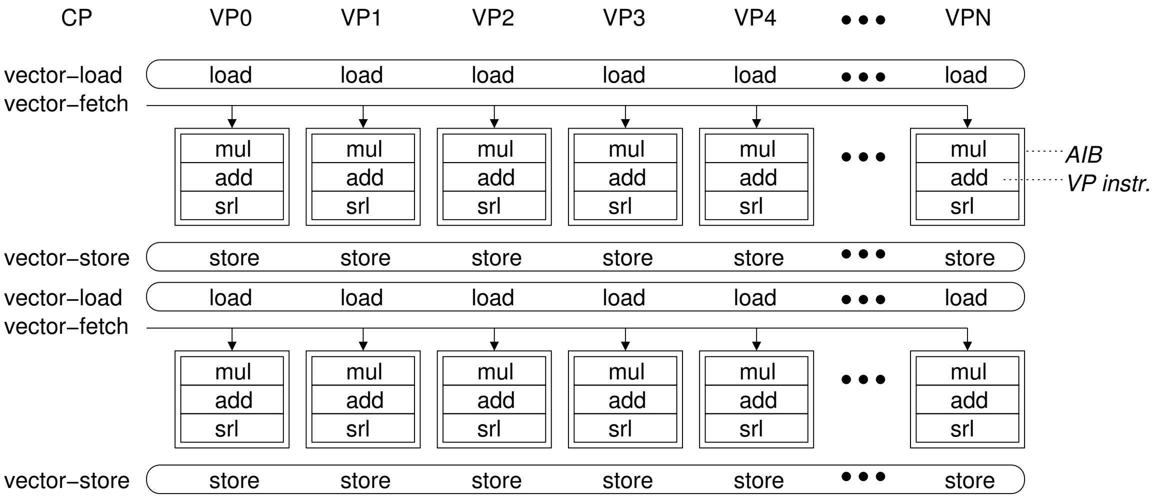


Figure 2-15: Example of vector code mapped to the VT architecture. Vector-load and vector-store commands read or write a register in each virtual processor. A vector-fetch sends a single AIB to each VP. An AIB is composed of simple virtual processor instructions which operate on each VP's local registers, here only the opcode is shown.

VT architecture, the VTU contains a parallel array of processing lanes as shown in Figure 2-18. As in a vector architecture, lanes are the physical processors onto which VPs are mapped. The physical register files in each lane are partitioned among the VPs, and the virtual processor vector is striped across the lane array. In contrast to traditional vector machines, the lanes in a VT machine execute decoupled from each other. Figure 2-19 shows an abstract view of how VP execution is time-multiplexed on the lanes for both vector-fetched and thread-fetched AIBs. This interleaving helps VT machines hide functional unit, memory, and thread-fetch latencies.

As shown in Figure 2-18, each lane contains both a *command management unit* (CMU) and an *execution cluster*. An execution cluster consists of a register file, functional units, and a small AIB cache. The CMU buffers commands from the control processor and holds pending thread-fetch commands from the lane's VPs. The CMU also holds the tags for the lane's AIB cache. The AIB cache can hold one or more AIBs and must be at least large enough to hold an AIB of the maximum size defined in the VT architecture. Typically the AIB cache is sized similarly to a small register file, for example Scale's AIB cache holds 32 instructions per cluster.

The CMU chooses either a vector-fetch or thread-fetch command to process. The fetch command contains an AIB address which is looked up in the AIB cache tags. If there is a miss, a request is sent to the AIB fill unit which retrieves the requested AIB from the primary cache. The fill unit handles one lane's AIB miss at a time, except if lanes are executing vector-fetch commands when refill overhead is amortized by broadcasting the AIB to all lanes simultaneously.

After a fetch command hits in the AIB cache or after a miss refill has been processed, the CMU generates an *execute directive* which contains an index into the AIB cache. For a vector-fetch command the execute directive indicates that the AIB should be executed by all VPs mapped to the lane, while for a thread-fetch command it identifies a single VP to execute the AIB. The execute directive is sent to a queue in the execution cluster, leaving the CMU free to begin processing the next command. The CMU is able to overlap the AIB cache refill for new fetch commands with the execution of previous ones, but it must track which AIBs have outstanding execute directives to avoid overwriting their entries in the AIB cache.

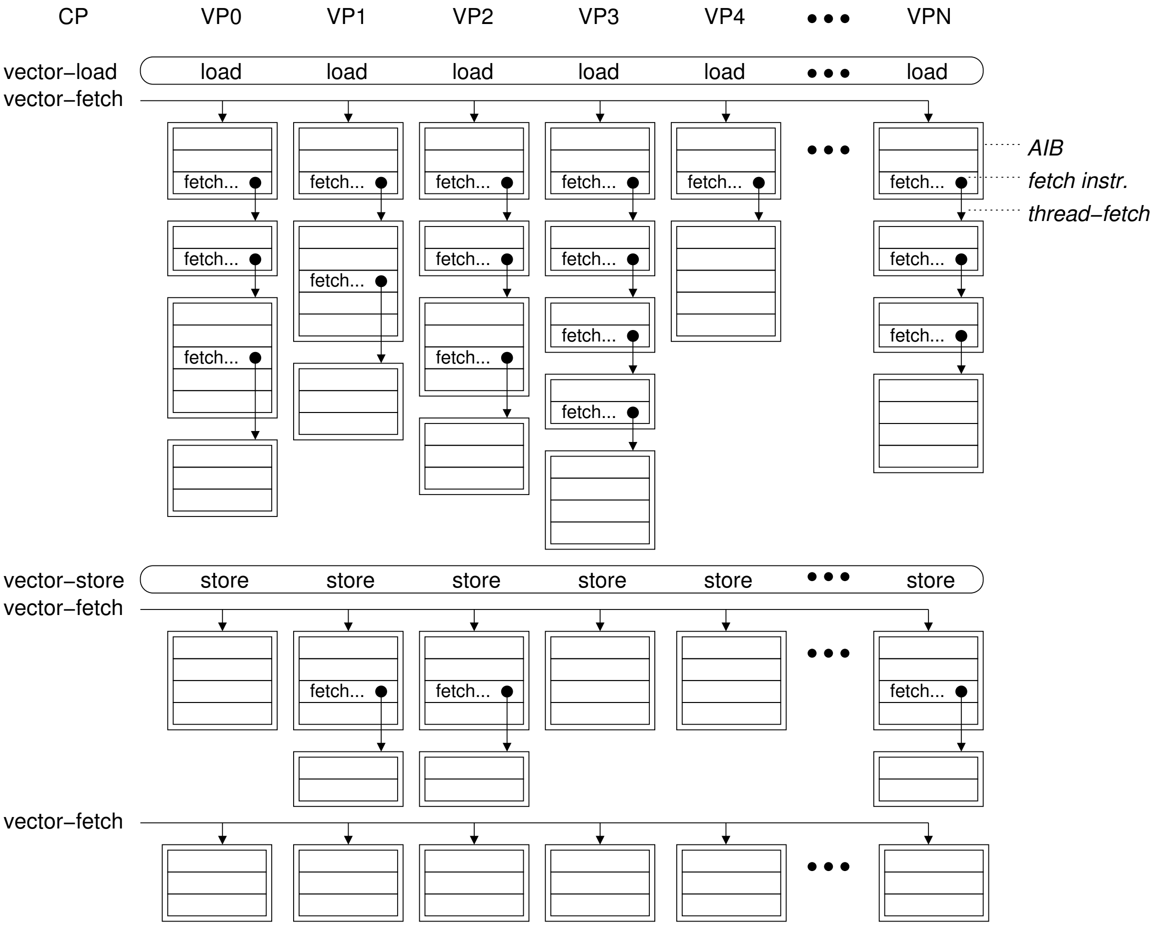


Figure 2-16: Example of vector-threaded code mapped to the VT architecture. A VP issues a thread-fetch to request its next AIB, otherwise the VP thread stops. Vector and threaded control, as well as vector memory accesses, may be interleaved at a fine granularity.

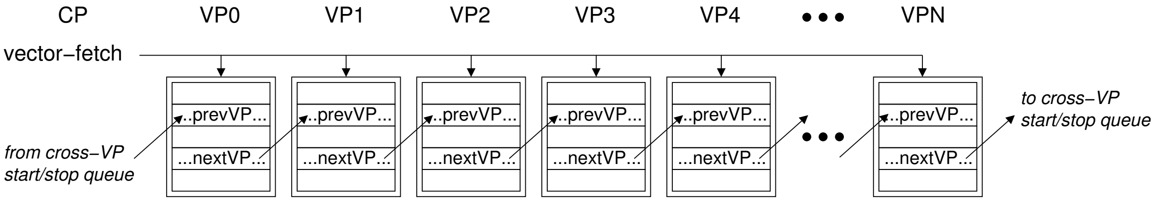


Figure 2-17: Example of cross-VP data transfers on the VT architecture. A vector-fetched AIB establishes a cross-VP chain that spans the virtual processor vector. The prevVP receives are encoded as instruction source operands, and the nextVP sends are encoded as instruction destinations. The prevVP data for VP0 comes from the cross-VP start/stop queue, and the final nextVP data is written to this same queue.

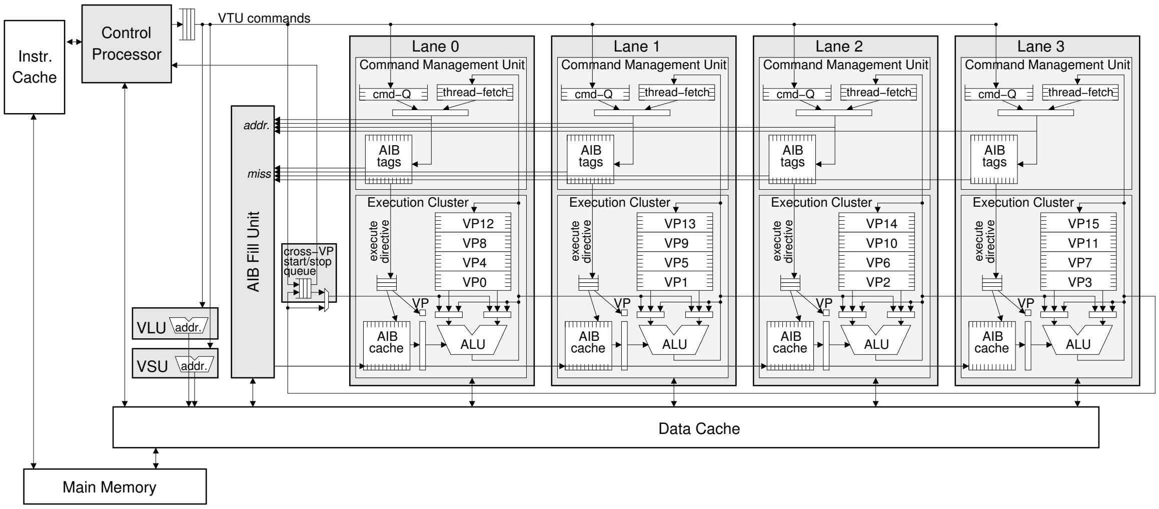


Figure 2-18: Vector-thread archetype.

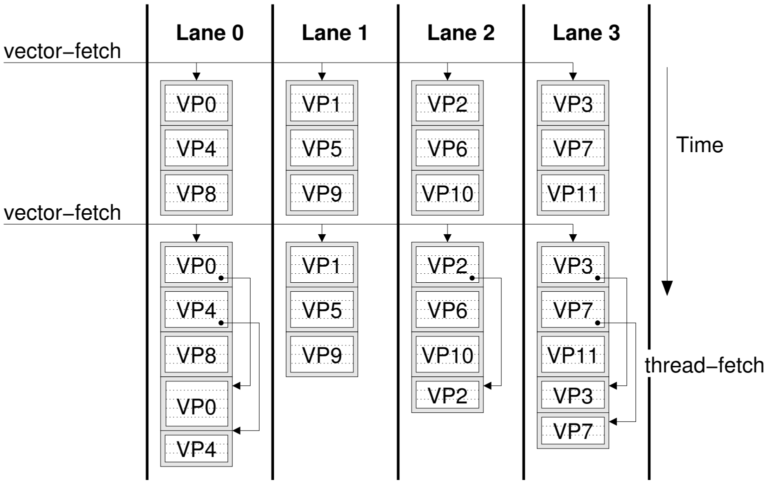


Figure 2-19: Time-multiplexing of VPs in a VT machine.

To process an execute directive, the execution cluster reads VP instructions one by one from the AIB cache and executes them for the appropriate VP. When processing an execute-directive from a vector-fetch command, all of the instructions in the AIB are executed for one VP before moving on to the next. The virtual register indices in the VP instructions are combined with the active VP number to create an index into the physical register file. To execute a fetch instruction, the execution cluster sends the thread-fetch request (i.e. the AIB address and the VP number) to the CMU.

The lanes in the VTU are inter-connected with a unidirectional ring network to implement the cross-VP data transfers. When an execution cluster encounters an instruction with a prevVP receive, it stalls until the data is available from its predecessor lane. When the VT architecture allows multiple cross-VP instructions in a single AIB, with some sends preceding some receives, the hardware implementation must provide sufficient buffering of send data to allow all the receives in an AIB to execute.

2.6 VT Semantics

For single-threaded architectures, program semantics are usually defined by a straightforward sequential interpretation of the linear instruction stream. A VT architecture, however, requires a more intricate abstract model since the instruction stream contains inherent concurrency. The problem is akin to defining the execution semantics of a multithreaded program.

The semantics of VT programs are defined relative to the operation of a VT virtual machine. This is essentially a functional version of the programmer's model in Figure 2-14. It includes a control processor and a vector of virtual processors that execute blocks of VP instructions. These processors operate concurrently in a shared memory address space.

A common programming pattern is for the control processor to vector-load input data, use a vector-fetch to launch VP threads, and then either vector-store output data or vector-load new input data. As expected, the VT semantics dictate that commands after a vector-fetch do not interfere with the execution of the VP threads, i.e. they happen "after" the vector-fetch. More precisely, *once a VP thread is launched, it executes to completion before processing a subsequent control processor command*.

VT semantics do however allow *different* VPs to process control processor commands "out of order" with respect to each other. For example, when there are back-to-back vector-fetch commands, some VP threads may execute the second command before others finish the first. Similarly, when a vector-store follows a vector-load, some VPs may execute the store before others complete the load. Traditional vector architectures have this same property.

Command interleaving concerns are mainly an issue for memory ordering semantics when the control processor and/or different VPs read and write the same memory locations. When necessary, memory dependencies between these processors are preserved via explicit memory fence and synchronization commands or atomic read-modify-write operations.

As a consequence of supporting multithreading, the semantics of a VT architecture are inherently nondeterministic. A VT program can potentially have a very large range of possible behaviors depending on the execution interleaving of the VP threads. As with any multithreaded architecture, it is the job of the programmer to restrict the nondeterminism in such a way that a program has well-defined behavior [Lee06]. Traditional multithreaded programs prune nondeterminism through memory-based synchronization primitives, and VT supports this technique for VP threads (via atomic memory instructions). However, this is often unnecessary since different VP threads do not usually read and write (or write and write) the same shared memory location before reaching one of the relatively frequent synchronization points introduced by commands from the

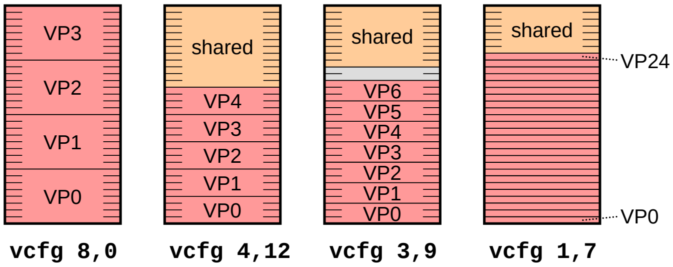


Figure 2-20: VP configuration with private and shared registers. The examples show various VP configurations mapped to a 32-entry register file. Each configuration command includes a private register count followed by a shared register count. Although only one register file is shown, the VPs would be striped across the lanes in a multi-lane VT machine.

control processor. Also, cross-VP data transfers provide a direct communication link between VPs without introducing nondeterminism.

It is important to note that nondeterminism is not unique to vector-thread architectures. Traditional vector architectures are also nondeterministic because they allow vector-loads, vector-stores, and scalar accesses to execute in parallel. The Cray X1 provides a variety of synchronization commands to order these different types of requests [cra03].

2.7 VT Design Space

As with any architectural paradigm, there is a large design space for VT architectures. The number of lanes in an implementation can scale to balance performance and complexity, and similar to a traditional vector architecture, this can even be done while retaining binary compatibility. Here, we also consider several extensions to VT which affect the programmer's model.

2.7.1 Shared Registers

In a VT architecture, the grouping of instructions into AIBs allows virtual processors to share temporary state. The general registers in each VP can be categorized as *private registers* (`pr`'s) and *shared registers* (`sr`'s). The main difference is that private registers preserve their values between AIBs, while shared registers may be overwritten by a different VP. In addition to holding temporary state, shared registers can hold scalar values and constants which are used by all VPs. The control processor can broadcast values by vector-writing the shared registers, but individual VP writes to shared registers are not coherent across lanes. That is to say, the shared registers are not global registers. A VT architecture may also expose shared *chain registers* (`cr`'s) which can be used to avoid register file reads and writes when communicating single-use temporaries between producer and consumer instructions [AHKW02].

2.7.2 VP Configuration

When software uses fewer private registers per VP, the physical register files in VT hardware can support a longer vector length (more VPs). To take advantage of this, a VT architecture may make the number of private and shared registers per VP configurable. Software on the control processor can change the VP configuration at a fine granularity, for example before entering each stripmined loop. The low-overhead configuration command may even be combined with setting the vector length. Figure 2-20 shows how various VP configurations map onto a register file in the VTU. Some traditional vector machines have also made the number of private registers per VP configurable, for example the Fujitsu VP100/VP200 [MU83].

2.7.3 Predication

In addition to conditional fetches, a VT instruction set may provide predication to evaluate small conditionals [HD86]. In such a design, each VP has a predicate register and VP instructions may execute conditionally under the predicate. Predication is a VP-centric view of the masked operations provided by many traditional vector instruction sets [SFS00].

2.7.4 Segment Vector Memory Accesses

In addition to the typical unit-stride and strided vector memory access patterns, VT architectures may provide *segment vector memory accesses*. Segments allow each VP to load or store several contiguous memory elements to support two-dimensional access patterns.

As an example, consider the strided vector accesses for loading RGB pixel values in Figure 2-9. Here, the instruction encoding has converted a two-dimensional access pattern into multiple one-dimensional access patterns. These multiple strided accesses obfuscate locality and can cause poor performance in the memory system due to increased bank conflicts, wasted address bandwidth, and additional access management state. A different data layout might enable the programmer to use more efficient unit-stride accesses, but reorganizing the data requires additional overhead and may be precluded by external API constraints.

Segment vector memory accesses more directly capture the two-dimensional nature of many higher-order access patterns. They are similar in spirit to the stream loads and stores found in stream processors [CEL$^+$03, KDK$^+$01]. Using segments, the RGB pixels can be loaded together with a command like:

```
vlbuseg vr1, rin, 3 # load R/G/B values to vr1/vr2/vr3
```

VP0.vr1 = mem[rin+0×3+0]
VP0.vr2 = mem[rin+0×3+1]
VP0.vr3 = mem[rin+0×3+2]
VP1.vr1 = mem[rin+1×3+0]
VP1.vr2 = mem[rin+1×3+1]
VP1.vr3 = mem[rin+1×3+2]
...
VPN.vr1 = mem[rin+N×3+0]
VPN.vr2 = mem[rin+N×3+1]
VPN.vr3 = mem[rin+N×3+2]

In this example, the stride between segments is equal to the segment size (3 bytes). A VT architecture may also support segment accesses with an arbitrary stride between segments. In this case, traditional strided accesses can be reduced to segment-strided accesses with a segment size of one element.

Segment accesses are useful for more than just array-of-structures access patterns; they are appropriate any time a programmer needs to move consecutive elements in memory into different vector registers. Segments can be used to efficiently access sub-matrices in a larger matrix or to implement vector loop-raking [ZB91].

2.7.5 Multiple-issue Lanes

A VT machine can improve performance by executing multiple operations per cycle in each lane. One approach is to issue multiple VP instructions per cycle without exposing the instruction-level parallelism (ILP) to software. This is analogous to superscalar execution for a VP, and it is similar in effect to the way in which multi-unit vector machines overlap the execution of multiple vector instructions.

Another ILP approach is to provide multiple execution clusters in the lanes and expose these to software as clustered resources within each VP, somewhat similar to a clustered VLIW interface. This is the approach adopted by Scale, and the reader may wish to compare Scale's programmer's model in Figure 4-1 to that for the baseline VT architecture in Figure 2-14. A lane in a multi-cluster VT machine has a single command management unit, and each cluster holds its own slice of the register file and the AIB cache. The properties of a VT architecture make it relatively simple to decouple the execution of different clusters with control and data queues, and the associated microarchitectural mechanisms are fully described in Chapter 5.

2.8 Vector-Threading

Applications can map to a vector-thread architecture in a variety of ways, but VT is especially well suited to executing loops. Under this model, each VP executes a single loop iteration and the control processor is responsible for managing the overall execution of the loop. For simple data-parallel loops, the control processor can use vector-fetch commands to send AIBs to all the VPs in parallel. For loops with cross-iteration dependencies, the control processor can vector-fetch an AIB that contains cross-VP data transfers. Thread-fetches allow the VPs to execute data-parallel loop iterations with conditionals or inner-loops. Vector memory commands can be used to exploit memory data-parallelism even in loops with non-data-parallel compute.

With VPs executing individual loop iterations, the multithreading in VT is very fine-grained—usually within a single function—and this local parallelism is managed by the compiler or the assembly language programmer. Compared to user-level threads that are managed by an operating system, VP threads in a VT architecture have orders of magnitude less overhead for initiation, synchronization, and communication. The control processor can simultaneously launch 10's–100's of VP threads with a single vector-fetch command, and it can likewise issue a barrier synchronization across the VPs using a single command. The VP threads benefit from shared data in the first-level cache, and when necessary the cache can be used for memory-based communication. VPs can also use cross-VP data transfers for low-overhead communication and synchronization.

When structured loop parallelism is not available, VPs can be used to exploit other types of thread parallelism. The primary thread running on the control processor can use individual VPs as *helper threads*. The control processor can assign small chunks of work to the VPs, or the VPs can help by prefetching data into the cache [CWT$^+$01, Luk01]. In another usage model, the control processor interaction is essentially eliminated and the VPs operate as *free-running threads*. This is a more traditional multithreaded programming model where the VP threads communicate through shared memory and synchronize using atomic memory operations. For example, the VPs may act as workers which process tasks from a shared work queue.

2.8.1 Vector Example: RGB-to-YCC

Figure 2-21 shows the RGB-to-YCC example coded for a generic vector-thread architecture. The control processor first configures the VPs with 6 private and 13 shared registers, and then vector-

```
RGBtoYCbCr:

  vcfg  6, 13 # configure VPs: 6 pr, 13 sr

  mult rinstripstr, vlmax, 3

  vwrsh sr0, CONST_Y_R
  vwrsh sr1, CONST_Y_G
  vwrsh sr2, CONST_Y_B
  vwrsh sr3, CONST_CB_R
  vwrsh sr4, CONST_CB_G
  vwrsh sr5, CONST_CB_B
  vwrsh sr6, CONST_CR_R
  vwrsh sr7, CONST_CR_G
  vwrsh sr8, CONST_CR_B
  vwrsh sr9, CONST_Y_RND
  vwrsh sr10,CONST_C_RND

stripmineloop:

  setvl vlen, n # (vl,vlen) = MIN(n,vlmax)

  vlbuseg pr0, rin, 3  # vector-load R/G/B
  vf      rgbycc_aib   # vector-fetch AIB
  vsb     pr7, routy   # vector-store Y
  vsb     pr8, routcb  # vector-store CB
  vsb     pr9, routcr  # vector-store CR

  add  rin, rin, rinstripstr # incr rin
  add  routy, routy, vlen    # incr routy
  add  routcb, routcb, vlen  # incr routcb
  add  routcr, routcr, vlen  # incr routcr
  sub  n, n, vlen            # decr n

  bnez n, stripmineloop      # stripmine
  jr   ra                    # func. return
```

```
rgbycc_aib:
  .aib begin

  mul sr11, pr0,  sr0  # R * Y_R
  mul sr12, pr1,  sr1  # G * Y_G
  add pr3,  sr11, sr12 # accum. Y
  mul sr11, pr2,  sr2  # B * Y_B
  add pr3,  pr3,  sr11 # accum. Y
  add pr3,  pr3,  sr9  # round Y
  srl pr3,  pr3,  16   # scale Y

  mul sr11, pr0,  sr3  # R * CB_R
  mul sr12, pr1,  sr4  # G * CB_G
  add pr4,  sr11, sr12 # accum. CB
  mul sr11, pr2,  sr5  # B * CB_B
  add pr4,  pr4,  sr11 # accum. CB
  add pr4,  pr4,  sr10 # round CB
  srl pr4,  pr4,  16   # scale CB

  mul sr11, pr0,  sr6  # R * CR_R
  mul sr12, pr1,  sr7  # G * CR_G
  add pr5,  sr11, sr12 # accum. CR
  mul sr11, pr2,  sr8  # B * CR_B
  add pr5,  pr5,  sr11 # accum. CR
  add pr5,  pr5,  sr10 # round CR
  srl pr5,  pr5,  16   # scale CR

  .aib end
```

Figure 2-21: RGB-to-YCC in generic vector-thread assembly code.

writes the constants for the color transformation to shared registers. The loop is stripmined as it would be for a traditional vector architecture. A segment vector-load fetches the RGB input data. The computation for one pixel transformation is organized as 21 VP instructions grouped into a single AIB. After vector-fetching this AIB, the control processor code uses three vector-stores to write the Y, Cb, and Cr output arrays.

Figure 2-22 diagrams how the RGB-to-YCC example executes on a VT machine. In contrast to the execution order on a vector machine (Figure 2-11), all of the instructions in the AIB are executed for one VP before moving on to the next. Although not shown in the example execution, chaining could allow the segment loads to overlap with the computation.

2.8.2 Threaded Example: Searching a Linked List

Figure 2-23 shows C code for a simple example loop that searches for many values in a linked list, and Figure 2-24 shows the example coded for a generic vector-thread architecture. The main loop is data-parallel, meaning that all of its iterations can execute concurrently. However, each iteration (each search) contains an inner while loop that executes for a variable number of iterations depending on its local search data. The loop is not vectorizable, but it can be mapped using VP threads. In Figure 2-24, a vector-fetch (for `search_aib`) launches the VP threads, and the VPs each use predicated fetch instructions to continue traversing the linked list until they find their search value. The VPs will each execute the inner while loop for a variable number of iterations, similar to

```
  CP       Lane 0    Lane 1    Lane 2    Lane 3

vlbuseg     0: lb     1: lb     2: lb     3: lb
            0: lb     1: lb     2: lb     3: lb
            0: lb     1: lb     2: lb     3: lb
            4: lb     5: lb     6: lb     7: lb
            4: lb     5: lb     6: lb     7: lb
            4: lb     5: lb     6: lb     7: lb
            8: lb     9: lb    10: lb    11: lb
            8: lb     9: lb    10: lb    11: lb
            8: lb     9: lb    10: lb    11: lb
           12: lb    13: lb    14: lb    15: lb
           12: lb    13: lb    14: lb    15: lb
           12: lb    13: lb    14: lb    15: lb
vf          0: *      1: *      2: *      3: *
            0: *      1: *      2: *      3: *
            0: +      1: +      2: +      3: +
            0: *      1: *      2: *      3: *
            0: +      1: +      2: +      3: +
            0: +      1: +      2: +      3: +
            0: >>     1: >>     2: >>     3: >>
            0: *      1: *      2: *      3: *
            0: *      1: *      2: *      3: *
            0: +      1: +      2: +      3: +
            0: *      1: *      2: *      3: *
            0: +      1: +      2: +      3: +
            0: +      1: +      2: +      3: +
            0: >>     1: >>     2: >>     3: >>
            0: *      1: *      2: *      3: *
            0: *      1: *      2: *      3: *
            0: +      1: +      2: +      3: +
            0: *      1: *      2: *      3: *
            0: +      1: +      2: +      3: +
            0: +      1: +      2: +      3: +
            0: >>     1: >>     2: >>     3: >>
            4: *      5: *      6: *      7: *
            4: *      5: *      6: *      7: *
            4: +      5: +      6: +      7: +
            4: *      5: *      6: *      7: *
            4: +      5: +      6: +      7: +
            4: +      5: +      6: +      7: +
            4: >>     5: >>     6: >>     7: >>
            [ ... ]
```

Figure 2-22: RGB-to-YCC vector-thread execution. Virtual processor numbers for the issuing element operations are indicated before the colon. Comparing the order of execution to the vector execution in Figure 2-11, a lane executes all of the operations in an AIB for one VP before moving on to the next.

```
void manySearch (int* in_ptr,
                 int* out_ptr,
                 Node* root,
                 int n) {
  for (int i=0; i<n; i++) {
    int searchv = *in_ptr++;
    Node* node = root;
    while(searchv != node->val)
      node = node->next;
    *out_ptr++ =  node->data;
  }
}
```

Figure 2-23: Linked list search example in C. The function searches a linked list for a set of values from an input array, and it writes the result data values from the searches to an output array.

```
manySearch:
  vcfg  3, 3 # configure VPs: 3 pr, 3 sr
  sll   rstripstr, vlmax, 2
  la    r1, search_aib
  vwrsh sr0, r1     # sr0 = search_aib
  vwrsh sr1, root   # sr1 = root
stripmineloop:
  setvl vlen, n # (vl,vlen) = MIN(n,vlmax)
  vlw   pr1, rin     # vector-load searchv
  vf    init_aib     # setup node pointers
  vf    search_aib   # launch searches
  vf    output_aib   # get result data
  vsw   pr2, rout    # vector-store output
  add  rin,  rin,  rstripstr # incr rin
  add  rout, rout, rstripstr # incr rout
  sub  n, n, vlen            # decr n
  bnez n, stripmineloop      # stripmine
  jr   ra                    # func. ret.
```

```
init_aib:
 .aib begin
 copy pr0, sr1   # node = root
 .aib end

search_aib:
 .aib begin
 lw  sr2, 4(pr0) # load node->val
 seq p, sr2, pr1 # p := (searchv == node->val)
 (!p) lw pr0, 8(pr0)# if !p: node = node->next
 (!p) fetch sr0     # if !p: fetch search_aib
 .aib end

output_aib:
 .aib begin
 lw pr2, 0(pr0)  # load node->data
 .aib end
```

Figure 2-24: Linked list search example in generic vector-thread assembly code. The VT mapping uses VP threads to perform searches in parallel.

the example in Figure 2-16. On a VT machine, the execution of the VP threads will be interleaved on the lanes, similar to the diagram in Figure 2-19. A final point is that, even though the search loop itself is not vectorizable, the code in Figure 2-24 still uses vector-loads to read the input data and vector-stores to write the output data.

Chapter 3

VT Compared to Other Architectures

In this chapter we compare the vector-thread architecture to other architectural paradigms. These paradigms can have many possible realizations, making it challenging to compare them in a systematic way. Since any software algorithm can map to any Turing-complete computer architecture, we judge different architectural paradigms on the following merits:

- Performance: Ideally the definition of the hardware-software interface should enable the implementation of a high-performance microarchitecture. Architectures can expose parallelism to allow many operations to execute simultaneously (e.g. the many element operations within a single vector instruction), and a simple instruction set interface can permit the use of low-latency hardware structures (e.g. the highly optimized components of a RISC pipeline).

- Efficiency: While attaining high performance, an implementation should minimize its consumption of critical resources including silicon area and power. In defining the interface between software and hardware, architectures strike a balance between the physical complexity of dynamic control algorithms implemented in hardware (e.g. register-renaming and dependency tracking in an out-of-order superscalar machine), and the overheads of static software control (e.g. storing and fetching wide instruction words in a VLIW machine). Architectures may also enable efficiency by exposing locality at the hardware/software interface to allow implementations to amortize control overheads across many operations.

- Programmability: An architecture is most useful when a variety of different applications can map to it efficiently. A clean software abstraction allows compilers or programmers to target an architecture with low effort. A more flexible architecture will allow more codes to obtain the benefits of high performance and efficiency. It may also be important for the software interface to provide scalability and portability across different microarchitectures implementing the same instruction set.

This chapter begins with a general analysis of loop-level parallelism, a primary target for VT architectures. We then compare VT to vector architectures, multiprocessors, out-of-order superscalars, VLIW architectures, and other research architectures. The discussion is qualitative, as quantitative comparisons are only possible for processor implementations. An advantage of an abstract comparison is that it applies to the general concepts rather than being limited to a particular implementation.

- Number of iterations:

  fixed-static: Known at compile time.

  fixed-dynamic: Known before entering loop.

  data-dep: Data-dependent exit condition.

- Cross-iteration dependencies:

  data-parallel: No cross-iteration dependencies.

  static: Statically analyzable dependencies.

  data-dep: Data-dependent, unanalyzable dependencies.

- Internal control flow:

  none: Straight-line code.

  cond-small: Small conditional expressions.

  cond-large: Large conditional blocks.

  inner-loop: Internal loop.

  unstructured: Various unstructured control flow.

- Inner-loop number of iterations:

  fixed-common-static: Same for all outer-loop iterations, known at compile time.

  fixed-common-dynamic: Same for all outer-loop iterations, known before entering loop.

  fixed-varies: Varies for different outer-loop iterations, but known before entering loop.

  data-dep: Data-dependent exit condition.

Figure 3-1: Loop categorization.

3.1 Loop-Level Parallelism

Architectures generally seek to improve performance or efficiency by exploiting the parallelism and locality inherent to applications. Program loops are ubiquitous in most applications, and provide an obvious source of instruction locality, as well as often providing data locality and parallelism between iterations. However, loops differ widely in their characteristics, and architectures are generally well-suited to only a subset of loop types.

Figure 3-1 presents a taxonomy for categorizing different types of loops based on properties which are important for exploiting loop-level parallelism. The loops in an application may have a nested structure; the taxonomy here applies to the loop which is chosen for parallelization. The first property is the number of iterations for which the loop executes. This may be fixed and known statically at compile time or dynamically before entering the loop, or it may be data-dependent and not known until the loop iterations actually execute. The second property involves data dependencies between loop iterations. Loops may have either no cross-iteration dependencies, cross-iteration dependencies which are statically known, or data-dependent cross-iteration dependencies (generally due to pointer aliasing or indexed array accesses) which can not be determined statically. The third property is the loop's internal control flow. Loop bodies may have either straight-line code with no control flow, small conditional if-then-else type statements, large conditional blocks which are infrequently executed, inner-loops, or unstructured procedure-call type control flow. Finally, for loops with inner-loops, the fourth property involves the number of iterations for which the inner-loop executes. This may be fixed and common across all the outer-loop iterations and known either statically or dynamically before entering the outer-loop, it may be fixed but vary for different outer-loop iterations, or it may be data-dependent and not known until the loop executes.

Vector architectures typically exploit data-parallel loops with no internal control flow and a fixed number of iterations. When a loop is vectorized, each of its instructions become multi-element operations which encode the parallel execution of many loop iterations (i.e. the vector length). A vector machine exploits loop-level parallelism by executing element operations concurrently, and locality by amortizing the control overhead. Vector architectures can not execute loops with cross-iteration dependencies except for, in some cases, a very limited set of reduction operations. Vector architectures generally can not handle loops with internal control flow or data-dependent exit conditions. However, with outer-loop vectorization they are able to parallelize loops which themselves contain inner-loops, as long as the inner-loops execute for a common number of iterations. Some vector architectures can also handle small conditional expressions by predicating the element operations. Predication is usually unsuitable for larger conditional blocks, and some vector architectures provide compress and expand operations which add an initial overhead but then allow a machine to execute only the non-predicated iterations [SFS00].

Multithreaded architectures can use individual threads to execute the iterations of a data-parallel loop. Compared to vector architectures, they are much more flexible and can parallelize loops with any type of internal control flow. However, multithreaded machines are not nearly as efficient as vector machines at exploiting loop-level instruction and data locality when executing vectorizable applications. Also, the synchronization costs between threads makes it impractical to parallelize loops that execute for a few number of iterations or that contain a small amount of work within each iteration. Typical multithreaded architectures can not parallelize loops with data-dependent cross-iteration dependencies. Some multithreaded architectures can parallelize loops with statically analyzable cross-iteration dependencies, but usually only by using relatively heavy-weight memory-based synchronization operations.

Superscalar architectures can reduce loop-level parallelism to instruction-level parallelism. In this way they can handle all types of loops, but they do not explicitly take advantage of the loops' parallelism or locality. They are particularly poor at exploiting parallelism when executing loops which contain unpredictable control flow, even when ample loop-level parallelism is available.

VLIW architectures typically use software pipelining to schedule the parallel execution of instructions across different loop iterations. In addition to vectorizable loops, this allows them to exploit parallelism in loops with statically analyzable cross-iteration dependencies. VLIW architectures can use predication to handle loops with data-dependent exit conditions or small conditional expressions, but they generally can not handle loops with other types of control flow. Although more flexible than vector machines, VLIW machines are not as efficient at executing vectorizable loops since they do not amortize control and data overheads.

The vector-thread architecture combines the advantages of vector and multithreaded architectures to efficiently exploit a wide variety of loop-level parallelism. At one extreme, when executing data-parallel loops, it is a vector architecture and can efficiently exploit the available parallelism and locality. However, it also includes mechanisms to handle loops with statically analyzable cross-iteration dependencies or internal control flow. At the other extreme, it is a multi-threaded architecture and can support many independent threads of control. The seamless transition between these modes of operation makes the vector-thread architecture unique in its ability to flexibly exploit parallelism.

3.2 VT versus Vector

The VT architecture builds upon earlier vector architectures [Rus78] by greatly enhancing programmability and flexibility. At the same time, VT attempts to allow implementations to retain most of the efficiencies of vector microprocessors [WAK$^+$96, KPP$^+$97, KTHK03].

Similar to a vector machine, a VT machine uses vector commands from a control processor to expose 100s of parallel operations. These operations execute over several machine cycles on a set of parallel lanes. In a vector processor, the instruction decoding and control overhead is typically done only once and amortized across the lanes. VT has more overhead since the lanes operate independently and the compute operations are held in AIB caches. Still, each lane performs only a single AIB cache tag-check, and the instructions within the AIB are then read using a short index into the small AIB cache. The control overhead is amortized over the multiple instructions within the AIB as well as the multiple VPs on each lane. Control overhead can be further amortized when an AIB has only one instruction (in a given cluster). In this case the instruction only needs to be read from the AIB cache and decoded once, and the datapath control signals can remain fixed as the instruction issues for all the VPs on the lane. Thus, when executing vectorizable code, VT machines are only marginally less efficient than vector machines at amortizing control overhead. This is especially true compared to modern Cray vector machines that are structured as a hierarchy of small two-lane vector units [D. 02, Faa98].

One of the most important benefits of vector processing is the ability to drive high bandwidth memory systems while tolerating long access latencies. The VT architecture retains traditional unit-stride and strided vector load and store commands. These expose memory access locality, allowing an implementation to efficiently fetch multiple elements with each access to the memory system. Furthermore, decoupling the memory access and register writeback portions of vector load commands (access/execute decoupling [EV96]) allows both vector and VT architectures to hide long memory access latencies. The machines can also exploit structured memory access patterns to initiate cache refills before the data is needed (refill/access decoupling [BKGA04]). For indexed loads and stores (gathers and scatters), the decoupled lanes of a VT machine help it to tolerate variation in the access latencies for individual elements. Other researches have studied run-ahead lanes to achieve the same benefit for traditional vector machines [LSCJ06].

The grouping of instructions into AIBs can give VT several advantages over vector architectures, even on vectorizable code. (1) Each vector-fetch encodes a larger amount of work, and this makes it easier for the control processor to run ahead of the VTU and process the scalar code. (2) Temporary state within an AIB can be held in shared registers within the register file, or even in chain registers to avoid register file accesses altogether. Reducing the per-VP private register requirements improves efficiency by increasing the maximum vector-length that the hardware can support. (3) AIBs also improve efficiency when used in conjunction with segment vector memory accesses. Although a traditional vector architecture could also provide segments, the multi-element load data writebacks and store data reads would leave idle pipeline slots when chaining with single-operation vector instructions. In VT, the multiple instructions within an AIB can keep the pipeline busy while data is transfered to or from memory.

One potential disadvantage of grouping instructions into AIBs is that there can be back-to-back dependencies between operations in the cluster execution pipeline. As a consequence, a VT machine must generally provide operand bypassing support within a cluster. These dependencies also preclude super-pipelining functional units such that basic operations have a latency greater than one cycle (for example, the Cray 1 took 3 cycles to perform an integer addition [Rus78]). However, it is not necessarily a limitation to base a machine's clock cycle time on a simple ALU with operand bypassing. This is a popular design point across many modern designs, as further pipelining tends to reduce power efficiency and greatly increase design complexity. Longer latency instructions like multiplies and floating point operations can be pipelined in a VT machine as they would in any other architecture, and simple instruction scheduling can avoid pipeline stalls.

3.3 VT versus Multiprocessors

VT's virtual processor threads are several orders of magnitude finer-grained than threads that typically run on existing multiprocessor architectures. In a traditional multithreaded program, threads tend to execute thousands, millions, or billions, of instructions between synchronization points. Even in integrated chip-multiprocessors (CMPs) and multi-core processors, memory-based communication and synchronization between threads can take 10s–100s of cycles [MMG$^+$06, KPP06]. In comparison, VT provides extremely fine-grained multithreading with very low overhead; a single vector-fetch command can launch, for example, 100 threads which each execute only 10 instructions. Vector-fetches and the cross-VP network provide low-cost synchronization, and a shared first-level cache enables low-overhead memory coherence.

The granularity of threading in VT enables a different type of parallelization. Traditional thread-level parallelism partitions the work of an application into separate chunks that execute concurrently, or it partitions a dataset to allow the threads to work on largely disjoint pieces. The parallelization, communication, and synchronization are usually explicitly managed by the application programmer. In VT, the thread parallelization is typically at the level of individual loop iterations within the compute kernels of an application. At this level, the memory dependencies between threads are often easy to analyze. A programmer targeting VT does not need to deal with many of the synchronization challenges that make multithreaded programming notoriously difficult. Even more promising, automatic loop-level threading is within the reach of modern compiler technology [BGGT01, CO03, LHDH05].

The amount of thread parallelism available in an application is often greater at a fine loop-level granularity than at a coarse task-level granularity [ACM88, CSY90, KDM$^+$98]. Although this may seem counter-intuitive at first, task-level threading can be hampered by memory dependencies that are either fundamental to the application or artificially introduced by the program mapping. For example, it is challenging to speed up an H.264 video decoder with thread parallelism due to dependencies both within and across frames [vdTJG03].

The vector-thread architecture exposes instruction and data locality to a far greater extent than traditional thread-level parallelization. When VT threads all execute iterations of the same loop, the bookkeeping overhead code is factored out and amortized by the control thread. The small AIB caches in VT hardware exploit locality between threads to provide high instruction bandwidth with low fetch energy. When there is data locality in the threads' memory access streams, vector loads and stores can provide high bandwidth efficiently. Furthermore, threads in a VT machine can reuse data in the shared first-level cache. Threads executing on multiprocessors can only share instructions and data in their second or third level caches, so these machines can not attain the same level of efficiency enabled by VT.

Although fine-grained vector-threading has its benefits, it also has drawbacks compared to coarse-grained multithreading in multiprocessors. A VT machine optimizes the case where threads are executing the same instructions, but performance may be limited by fetch bandwidth if the threads are all following different code paths. A VT architecture also has inherently limited scalability due to the complexity of the interconnection between the lanes and the shared cache. Instead of scaling a single VT processor to provide more compute power, the VT architecture should be used to build a flexible and efficient processor core. Larger systems can be constructed as VT-multiprocessors with a traditional multithreaded programming model across the cores. This is the same approach that has been commonly used in the design of vector supercomputers [ABHS89, HUYK04].

3.4 VT versus Out-of-Order Superscalars

Superscalar architectures retain a simple RISC programming model and use hardware to dynamically extract instruction-level parallelism. Instructions are staged in a reorder buffer which renames registers and tracks dependencies to enable out-of-order execution. Superscalar machines invest heavily in branch prediction to run ahead and feed speculative instructions into the reorder buffer, and the machines must implement intricate repair mechanisms to recover from misspeculations. When executing long-latency instructions like floating-point operations or loads which miss in the cache, superscalar machines rely on large instruction windows to buffer dependent operations and find instructions which can execute in parallel.

Superscalar processors primarily root out local parallelism in the instruction stream. They can also exploit loop-level parallelism, but in order to do this they must buffer all of the instructions in one iteration in order to get to the next. This becomes especially difficult when a loop contains even simple control flow, like an if-then-else statement. The superscalar processor must correctly predict the direction of all branches within the loop iteration in order to get to the next iteration, even though the execution of the next iteration does not depend on these local branches.

The vector-thread architecture has the advantage of exposing loop-level parallelism in the instruction set encoding. The virtual processors execute loop iterations in parallel, and they each process the local control flow independently. It is easily possible for 100 virtual processors to simultaneously execute a loop iteration which contains around 100 instructions. This is equivalent to an instruction window of 10,000, at least in the sense that instructions which would be separated by this distance in a scalar execution can execute concurrently. This huge window of parallelism is achieved by VT with no instruction buffering, no dynamic dependency tracking, and no speculation.

Although ILP is costly to exploit on a grand scale, it can provide an important low-cost performance boost when applied at a local level. Given that each lane in a VT machine has one load/store port into the memory system, it makes sense for a balanced machine to execute several instructions per cycle within each lane. Scale supports ILP with multi-cluster lanes that are exposed to software as clustered VPs.

Superscalars are well suited for executing code that lacks structured parallelism and locality. Aside from instruction and data caching, extracting local instruction-level parallelism is one of the only ways to speed up such code, and this is the forte of superscalar architectures. Applying this observation to VT, an implementation could use a superscalar architecture for the control processor. In such a design, the vector-thread unit offloads the structured portions of applications, leaving the superscalar to concentrate its efforts on the more difficult ILP extraction. Similar observations have also been made about hybrid vector/superscalar architectures [EAE$^+$02, QCEV99, VJM99].

3.5 VT versus VLIW

VLIW architectures exploit instruction-level parallelism with a simple hardware/software interface. Whereas superscalars extract ILP dynamically in hardware, VLIW machines expose ILP to software. Wide instruction words control a parallel set of functional units on a cycle-by-cycle basis. Although simple, this low-level interface does not expose locality in the instruction or data streams. Compared to a vector or vector-thread machine, a VLIW implementation is less able to amortize instruction control overheads. Not only must the hardware fetch and issue instructions every cycle, but unused issue slots are encoded as nops (no-operations) that still consume resources in the machine. For data access, VLIW machines use regular scalar loads and stores which are inefficient compared to vector accesses.

VLIW architectures are often used in conjunction with software pipelining to extract parallelism from loops with cross-iteration dependencies. A vector-thread architecture can also parallelize such loops, but it does so without using software pipelining. Instead, the vector-thread code must only schedule instructions for one loop iteration mapped to one virtual processor, and cross-iteration dependencies are directly encoded as prevVP receives and nextVP sends. The available parallelism is extracted at run time as the VPs execute on adjacent lanes. Dynamic hardware scheduling of the explicit cross-VP data transfers allows the execution to automatically adapt to the software critical path. Whereas a statically scheduled VLIW mapping has trouble handling unknown or unpredictable functional unit and memory access latencies, the decoupled lanes in a vector-thread machine simply stall (or execute independent instructions) until the data is available. In addition to improving performance, this dynamic hardware scheduling enables more portable and compact code.

3.6 VT versus Other Architectures

Speculative Threading. Multiscalar processors [SBV95] can flexibly exploit arbitrary instruction-level and loop-level parallelism. The architecture includes a centralized sequencer which traverses the program control flow graph and assigns tasks to independent processing units. A task can correspond to a loop iteration, but this is not a requirement. The processing units are connected in a ring to allow data dependencies to be forwarded from one to the next, i.e. from older tasks to newer tasks. Similar to VT's cross-VP data transfers, a compiler for Multiscalar determines these register dependencies statically. Multiscalar uses a fundamentally speculative parallelization model. It predicts when tasks can execute concurrently, and it uses an address resolution buffer to detect conflicts and undo invalid stores when incorrectly speculated tasks are squashed.

The superthreaded architecture [THA+99] pipelines threads on parallel execution units in a similar manner as Multiscalar. Instead of using a centralized controller to spawn threads, however, the architecture allows each thread to explicitly, and often speculatively, fork its successor. There is always a single head thread, and when a misspeculation is detected all of the successor threads are squashed. The main difference between the superthreaded architecture and Multiscalar is the hardware used to handle data speculation. The superthreaded architecture uses a simpler data buffer, and software is responsible for explicitly managing memory ordering and detecting violations.

Like Multiscalar and the superthreaded architecture, VT exposes distant parallelism by allowing a control thread to traverse a program and launch parallel sub-threads. VT also uses a ring network to pass cross-iteration values between the threads. However, since VT is targeted specifically at parallelizing loop iterations, it is able to dispatch multiple iterations simultaneously (with a vector-fetch) and to exploit instruction reuse between the threads. Loop parallelization also allows VT to provide efficient vector memory accesses. Instead of executing one task per processing unit, VT uses multithreading of virtual processors within a lane to improve utilization and efficiency. In contrast to these speculative threading architectures, VT primarily exploits non-speculative structured parallelism. Multiscalar and the superthreaded architecture can extract parallelism from a wider range of code, but VT can execute many common parallel loop types with simpler logic and less software overhead.

Micro-Threading. Jesshope's micro-threading [Jes01] parallelizes loops in a similar manner as VT. The architecture uses a command analogous to a vector-fetch to launch micro-threads which each execute one loop iteration. Micro-threads execute on a parallel set of pipelines, and multithreading is used to hide load and branch latencies. In contrast to VT's point-to-point cross-VP data transfers, communication between threads is done through a global register file which maintains full/empty bits for synchronization.

Multi-Threaded Vectorization. Chiueh's multi-threaded vectorization [Tzi91] extends a vector machine to handle loop-carried dependencies. The architecture adds an inter-element issue period parameter for vector instructions. To parallelize loops with cross-iteration dependencies, the compiler first software-pipelines a loop. Then it maps each loop operation to a vector instruction by determining its issue time and issue period. As a result, the approach is limited to a single lane and requires the compiler to have detailed knowledge of all functional unit latencies. In comparison, VT groups loop instructions into an AIB with explicit cross-iteration sends and receives. AIBs execute on parallel lanes, and the execution dynamically adapts to runtime latencies.

Partitioned Vector. Vector lane threading [RSOK06] targets applications that do not have enough data-level parallelism to keep the lanes in a vector unit busy. Multiple control threads each operate on a configurable subset of the lanes in a vector unit. For example, two control threads could each use four lanes in an eight-lane vector machine. The control threads either run on independent control processors or they run concurrently on an SMT control processor. In contrast to VT, the lanes themselves (and the VPs) use regular vector control, and multithreading is provided by the control processor(s).

The vector lane threading proposal also evaluates adding 4 KB caches and instruction sequencers to allow each lane to operate as an independent scalar processor. In this mode, the architecture operates as a regular chip multiprocessor. This is different than VT since there is no longer a control processor issuing vector commands to the lanes. Furthermore, in this mode each lane runs only one thread, compared to multithreading multiple VPs.

Partitioned VLIW. The Multiflow Trace/500 architecture [CHJ$^+$90] contains two clusters with 14 functional units each. It can operate as a 28-wide VLIW processor, or it can be partitioned into two 14-wide processors. The XIMD architecture [WS91] generalizes this concept by giving each functional unit its own instruction sequencer. In this way it achieves the flexibility of multithreading, and it can parallelize loop iterations that have internal control flow or cross-iteration dependencies [NHS93]. XIMD software can also run the threads in lock-step to emulate a traditional single-threaded VLIW machine.

A primary difference between XIMD and VT is that XIMD's threads share a large global register file (which has 24 ports in the hardware prototype). In comparison, the VT abstraction gives virtual processors their own private registers. This provides isolation between threads and exposes operand locality within a thread. As a result, hardware implementations of VT are able to parallelize VP execution across lanes (with independent register files), as well as time-multiplex multiple VPs on a single lane. The explicit cross-VP data transfers in a VT architecture enable decoupling between the lanes (and between the VP threads).

The distributed VLIW (DVLIW) architecture [ZFMS05] applies the partitioned VLIW concept to a clustered machine. It proposes a distributed program counter and branching mechanism that together allow a single thread's instructions to be organized (and compressed) independently for each cluster. The Voltron architecture [ZLM07] extends this work and adds a decoupled mode of operation that it uses to exploit thread parallelism. Voltron essentially augments a chip multiprocessor with a scalar operand network and a program control network. In comparison, VT exploits fine-grained thread parallelism within a processor core. This allows it to reduce inter-thread communication and synchronization overheads, and to amortize control and data overheads across threads.

Hybrid SIMD/MIMD. Several older computer systems have used hybrid SIMD/MIMD architectures to merge vector and threaded computing, including TRAC [SUK$^+$80], PASM [SSK$^+$81], Opsila [BA93], and Triton [PWT$^+$94]. The level of integration in these architectures is coarser than in VT. They are organized as collections of processors that either run independently or under common control, and the machines must explicitly switch between these separate MIMD and SIMD operating modes. Unlike VT, they do not amortize memory access overheads with vector load and store instructions, and they do not amortize control overheads by mapping multiple virtual processors to each physical processor.

Raw. The Raw microprocessor [TKM$^+$02] connects an array of processing tiles with a scalar operand network [TLAA05]. Scalar operands are primarily encountered when extracting local instruction-level parallelism. They also appear as cross-iteration values during loop-level parallelization. Raw uses simple single-issue processor tiles and parallelizes scalar code across multiple tiles. Inter-tile communication is controlled by programmed switch processors and must be statically scheduled to tolerate latencies.

The vector-thread architecture isolates scalar operand communication within a single highly-parallel processor core. Inter-cluster data transfers provide low-overhead scalar communication, either across lanes for cross-VP data transfers, or within a lane in a multi-cluster VT processor. These scalar data transfers are statically orchestrated, but decoupling in the hardware allows execution to dynamically adapt to runtime latencies. Like Raw, VT processors can provide zero-cycle latency and occupancy for scalar communication between functional units [TLAA05], and the inter-cluster network in a VT machine provides even lower overhead than Raw's inter-tile network.

Imagine. The Imagine stream processor [RDK$^+$98] is similar to vector machines, with the main enhancement being the addition of stream load and store instructions that pack and unpack arrays of multi-field records stored in DRAM into multiple vector registers, one per field. In comparison, the vector-thread architecture uses a conventional cache to enable unit-stride transfers from DRAM, and provides segment vector memory commands to transfer arrays of multi-field records between the cache and VP registers.

Like VT, Imagine also improves register file locality compared with traditional vector machines by executing all operations for one loop iteration before moving to the next. However, Imagine instructions use a low-level VLIW ISA that exposes machine details such as the number of physical registers and lanes. VT provides a higher-level abstraction based on VPs and AIBs which allows the same binary code to run on VT implementations with differing numbers of lanes and physical registers.

Polymorphic Architectures. The Smart Memories [MPJ$^+$00] project has developed an architecture with configurable processing tiles which support different types of parallelism, but it has different instruction sets for each type and requires a reconfiguration step to switch modes. The TRIPS processor [SNL$^+$03] similarly must explicitly *morph* between instruction, thread, and data parallelism modes. These mode switches limit the ability to exploit multiple forms of parallelism at a fine-granularity, in contrast to VT which seamlessly combines vector and threaded execution while also exploiting local instruction-level parallelism.

Chapter 4

Scale VT Instruction Set Architecture

The Scale architecture is an instance of the vector-thread architectural paradigm. As a concrete architecture, it refines the VT abstract model and pins down all the details necessary for an actual hardware/software interface. Scale is particularly targeted at embedded systems—the goal is to provide high performance at low power for a wide range of applications while using only a small area. This chapter presents the programmer's model for Scale and discusses some design decisions.

4.1 Overview

Scale builds on the generic VT architecture described in Section 2.5, and Figure 4-1 shows an overview of the Scale programmer's model. The control processor (CP) is a basic 32-bit RISC architecture extended with vector-thread unit (VTU) commands. The CP instruction set is loosely based on MIPS-II [KH92]. The hardware resources in the VTU are abstracted as a configurable vector of virtual processors. Scale's VPs have clustered register files and execution resources. They execute RISC-like VP instructions which are grouped into atomic instruction blocks. The control processor may use a vector-fetch to send an AIB to all of the VPs en masse, or each VP may issue thread-fetches to request its own AIBs. The control processor and the VPs operate concurrently in a shared memory address space.

A summary of the VTU commands is shown in Tables 4.1 and 4.2. The `vcfgvl` command configures the VPs' register requirements—which, in turn, determines the maximum vector length (`vlmax`)—and sets the active vector length (`vl`). The vector-fetch and VP-fetch commands send AIBs to either all of the active VPs or to one specific VP. Three types of synchronization commands (`vsync`, `vfence`, and `vpsync`) allow the control processor to constrain the ordering of memory accesses between itself and the VPs. System code can reset the VTU with the `vkill` command to handle exceptional events. The `vwrsh` command lets the control processor distribute a shared data value to all the VPs, and the `vprd` and `vpwr` commands allow it to read and write individual VP registers. The control processor can provide initial cross-VP data and consume the final results using the `xvppush`, `xvppop`, and `xvpdrop` commands. A variety of vector-load and vector-store commands support unit-stride and segment-strided memory access patterns, and a shared vector-load fetches a single data value for all the VPs. All of the VTU commands are encoded as individual control processor instructions, except for the `vcfgvl` command which the assembler may need to expand into two or three instructions.

The following sections describe the virtual processors and the VTU commands in more detail.

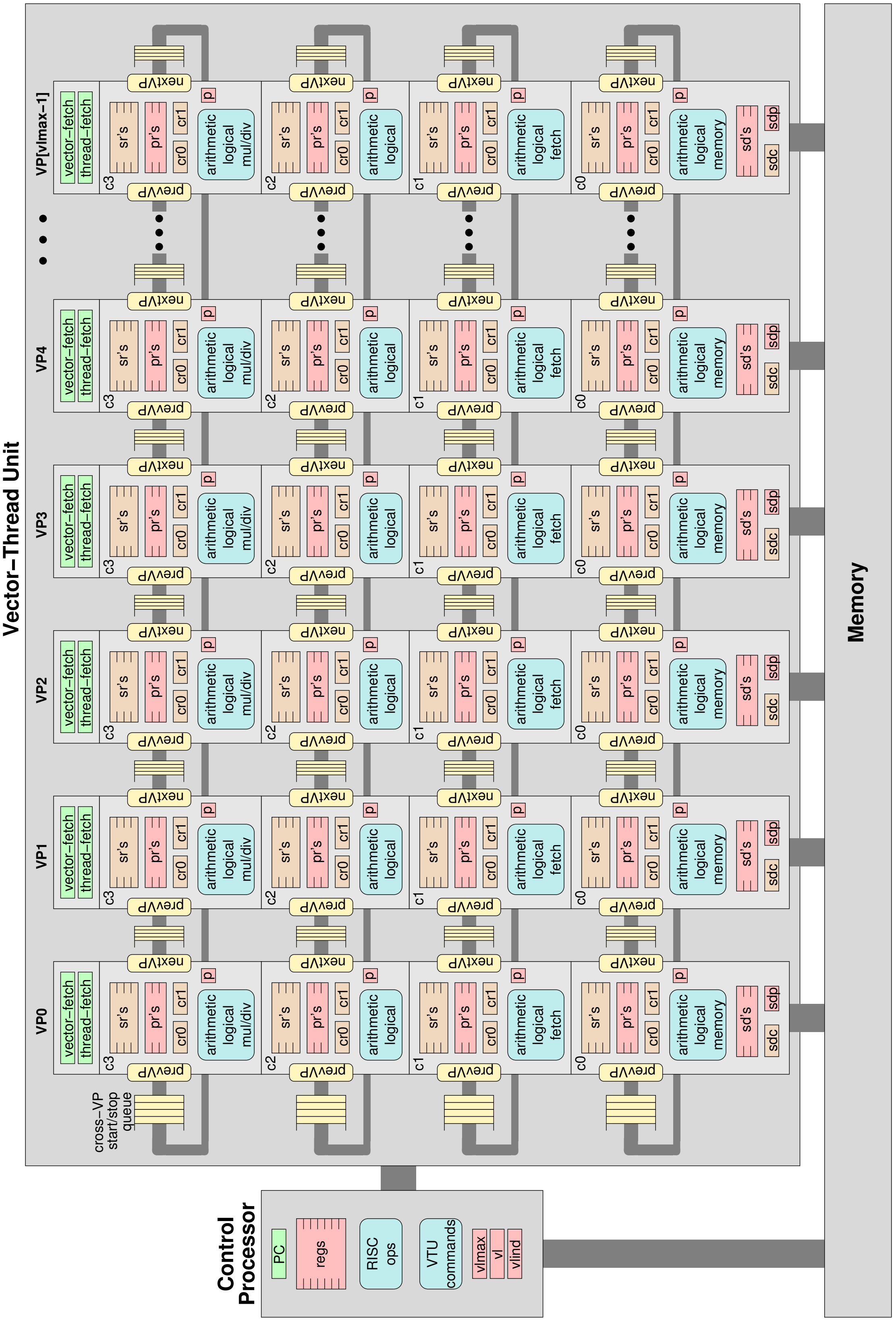


Figure 4-1: Programmer's model for the Scale VT architecture.

Operation	***Assembly format***	***Summary***
configure VPs and set vector length	`vcfgvl` $\mathbf{r_{dst}}, \mathbf{r_{len}}, \mathbf{n_{c0p}}, \mathbf{n_{c0s}},$ $\mathbf{n_{c1p}}, \mathbf{n_{c1s}}, \mathbf{n_{c2p}}, \mathbf{n_{c2s}}, \mathbf{n_{c3p}}, \mathbf{n_{c3s}}$	Configure the VPs with $\mathbf{n_{c0p}}$ private and $\mathbf{n_{c0s}}$ shared registers in cluster 0, $\mathbf{n_{c1p}}$ private and $\mathbf{n_{c1s}}$ shared registers in cluster 1, etc. The $\mathbf{n_{c0p}}$ parameter is also used as the number of private store-data registers in cluster 0. State in the VPs becomes undefined if the new configuration is not the same as the existing configuration. The configuration determines the maximum vector length which the VTU hardware can support, and this value is written to `vlmax`. The new vector length is then computed as the minimum of $\mathbf{r_{len}}$ and `vlmax`, and this value is written to `vl` and $\mathbf{r_{dst}}$.
set vector length	`setvl` $\mathbf{r_{dst}}, \mathbf{r_{len}}$	The new vector length length is computed as the minimum of $\mathbf{r_{len}}$ and `vlmax`, and this value is written to `vl` and $\mathbf{r_{dst}}$.
vector-fetch	`vf` **label**	Send the AIB located at the label to every active VP in the VPV.
VP fetch	`vpf`[.nb] $\mathbf{r_{vp}}$, **label**	Send the AIB located at the label to the VP specified by $\mathbf{r_{vp}}$. The *.nb* version is non-blocking.
vector sync	`vsync`	Stall until every VP in the VPV is idle and has no outstanding memory operations.
vector fence	`vfence`	Complete all memory operations from previous vector commands (vector-fetch, vector-load, and vector-store) before those from any subsequent vector commands.
VP sync	`vpsync` $\mathbf{r_{vp}}$	Stall until the VP specified by $\mathbf{r_{vp}}$ is idle and has no outstanding memory operations.
VTU kill	`vkill`	Kill any previously issued VTU commands at an arbitrary point of partial execution and reset the VTU into an idle state.
VP reg-read	`vprd`[.nb] $\mathbf{r_{vp}}, \mathbf{r_{dst}}, \mathbf{c_{src}}/\mathbf{r_{src}}$	Copy $\mathbf{c_{src}}/\mathbf{r_{src}}$ in the VP specified by $\mathbf{r_{vp}}$ to $\mathbf{r_{dst}}$. The *.nb* version is non-blocking.
VP reg-write	`vpwr`[.nb] $\mathbf{r_{vp}}, \mathbf{r_{src}}, \mathbf{c_{dst}}/\mathbf{r_{dst}}$	Copy $\mathbf{r_{src}}$ to $\mathbf{c_{dst}}/\mathbf{r_{dst}}$ in the VP specified by $\mathbf{r_{vp}}$. The *.nb* version is non-blocking.
vector reg-write shared	`vwrsh` $\mathbf{r_{src}}, \mathbf{c_{dst}}/\mathbf{r_{dst}}$	Copy $\mathbf{r_{src}}$ to $\mathbf{c_{dst}}/\mathbf{r_{dst}}$ in every VP in the VPV. $\mathbf{r_{dst}}$ must be a shared register.
cross-VP push	`xvppush` $\mathbf{r_{src}}, \mathbf{c_{dst}}$	Push a copy of $\mathbf{r_{src}}$ to the cross-VP start/stop queue for cluster $\mathbf{c_{dst}}$.
cross-VP pop	`xvppop` $\mathbf{r_{dst}}, \mathbf{c_{src}}$	Pop from the cross-VP start/stop queue for cluster $\mathbf{c_{src}}$ and store the value into $\mathbf{r_{dst}}$.
cross-VP drop	`xvpdrop` $\mathbf{c_{src}}$	Pop from the cross-VP start/stop queue for cluster $\mathbf{c_{src}}$ and discard the value.

Table 4.1: Basic VTU commands.

Operation	***Assembly format***	***Summary***
unit-stride vector load	$\texttt{v}\mathcal{L}\ \mathbf{r_{base}}, \mathbf{c_{dst}}/\mathbf{r_{dst}}$ $\texttt{v}\mathcal{L}\texttt{ai}\ \mathbf{r_{base}}, \mathbf{r_{inc}}, \mathbf{c_{dst}}/\mathbf{r_{dst}}$	Each active VP in the VPV loads the element with address: $\mathbf{r_{base}} + \mathbf{width} \cdot \mathbf{VPindex}$ (where $\mathbf{width}$ is the number of bytes determined by the opcode, and $\mathbf{VPindex}$ is the VP's index number) to $\mathbf{c_{dst}}/\mathbf{r_{dst}}$. The load-data is zero-extended for the `u` versions of the opcodes, or sign-extended otherwise. $\mathbf{r_{dst}}$ must be a private register. For the *ai* versions of the opcode, $\mathbf{r_{base}}$ is automatically incremented by $\mathbf{r_{inc}}$.
segment-strided vector load	$\texttt{v}\mathcal{L}\texttt{segst}\ \mathbf{n}, \mathbf{r_{base}}, \mathbf{r_{str}}, \mathbf{c_{dst}}/\mathbf{r_{dst}}$ $\texttt{v}\mathcal{L}\texttt{seg}\ \mathbf{n}, \mathbf{r_{base}}, \mathbf{c_{dst}}/\mathbf{r_{dst}}$ $\texttt{v}\mathcal{L}\texttt{st}\ \mathbf{r_{base}}, \mathbf{r_{str}}, \mathbf{c_{dst}}/\mathbf{r_{dst}}$	Similar to *unit-stride vector load*, except each VP loads $\mathbf{n}$ elements with addresses: $\mathbf{r_{base}} + \mathbf{r_{str}} \cdot \mathbf{VPindex} + \mathbf{width} \cdot 0$ $\mathbf{r_{base}} + \mathbf{r_{str}} \cdot \mathbf{VPindex} + \mathbf{width} \cdot 1$ $\ldots$ $\mathbf{r_{base}} + \mathbf{r_{str}} \cdot \mathbf{VPindex} + \mathbf{width} \cdot (\mathbf{n} - 1)$ to $\mathbf{c_{dst}}/(\mathbf{r_{dst}}{+}0)$, $\mathbf{c_{dst}}/(\mathbf{r_{dst}}{+}1)$, $\ldots$, $\mathbf{c_{dst}}/(\mathbf{r_{dst}}{+}(\mathbf{n}{-}1))$. For the simplified *seg* versions of the opcode, the stride ($\mathbf{r_{str}}$) is equal to the segment width ($\mathbf{width} \cdot \mathbf{n}$). For the simplified *st* versions of the opcode, the segment size ($\mathbf{n}$) is 1.
shared vector load	$\texttt{v}\mathcal{L}\texttt{sh}\ \mathbf{r_{base}}, \mathbf{c_{dst}}/\mathbf{r_{dst}}$	Every VP in the VPV loads the element with address $\mathbf{r_{base}}$ to $\mathbf{c_{dst}}/\mathbf{r_{dst}}$. The load-data is zero-extended for the `u` versions of the opcodes, or sign-extended otherwise. $\mathbf{r_{dst}}$ must be a shared or chain register.
unit-stride vector store	$\texttt{v}\mathcal{S}\ \mathbf{r_{base}}, \texttt{c0}/\mathbf{sd_{src}}$ $\texttt{v}\mathcal{S}\ \mathbf{r_{base}}, \texttt{c0}/\mathbf{sd_{src}}, \mathcal{P}$ $\texttt{v}\mathcal{S}\texttt{ai}\ \mathbf{r_{base}}, \mathbf{r_{inc}}, \texttt{c0}/\mathbf{sd_{src}}$ $\texttt{v}\mathcal{S}\texttt{ai}\ \mathbf{r_{base}}, \mathbf{r_{inc}}, \texttt{c0}/\mathbf{sd_{src}}, \mathcal{P}$	Each active VP in the VPV stores the element in `c0`/$\mathbf{sd_{src}}$ to the address: $\mathbf{r_{base}} + \mathbf{width} \cdot \mathbf{VPindex}$ (where $\mathbf{width}$ is the number of bytes determined by the opcode, and $\mathbf{VPindex}$ is the VP's index number). For the predicated versions, the store only occurs if the store-data predicate register in cluster 0 is set to one with the $(\texttt{c0}/\texttt{sdp})$ argument or zero with the $(!\texttt{c0}/\texttt{sdp})$ argument. $\mathbf{sd_{src}}$ must be a private store-data register. For the *ai* versions of the opcode, $\mathbf{r_{base}}$ is automatically incremented by $\mathbf{r_{inc}}$.
segment-strided vector store	$\texttt{v}\mathcal{S}\texttt{segst}\ \mathbf{n}, \mathbf{r_{base}}, \mathbf{r_{str}}, \texttt{c0}/\mathbf{sd_{src}}$ $\texttt{v}\mathcal{S}\texttt{segst}\ \mathbf{n}, \mathbf{r_{base}}, \mathbf{r_{str}}, \texttt{c0}/\mathbf{sd_{src}}, \mathcal{P}$ $\texttt{v}\mathcal{S}\texttt{seg}\ \mathbf{n}, \mathbf{r_{base}}, \texttt{c0}/\mathbf{sd_{src}}$ $\texttt{v}\mathcal{S}\texttt{seg}\ \mathbf{n}, \mathbf{r_{base}}, \texttt{c0}/\mathbf{sd_{src}}, \mathcal{P}$ $\texttt{v}\mathcal{S}\texttt{st}\ \mathbf{r_{base}}, \mathbf{r_{str}}, \texttt{c0}/\mathbf{sd_{src}}$ $\texttt{v}\mathcal{S}\texttt{st}\ \mathbf{r_{base}}, \mathbf{r_{str}}, \texttt{c0}/\mathbf{sd_{src}}, \mathcal{P}$	Similar to *unit-stride vector store*, except each VP stores the $\mathbf{n}$ elements in `c0`/$(\mathbf{sd_{src}}+0)$, `c0`/$(\mathbf{sd_{src}}+1)$, $\ldots$, `c0`/$(\mathbf{sd_{src}}+(\mathbf{n}-1))$ to the addresses: $\mathbf{r_{base}} + \mathbf{r_{str}} \cdot \mathbf{VPindex} + \mathbf{width} \cdot 0$ $\mathbf{r_{base}} + \mathbf{r_{str}} \cdot \mathbf{VPindex} + \mathbf{width} \cdot 1$ $\ldots$ $\mathbf{r_{base}} + \mathbf{r_{str}} \cdot \mathbf{VPindex} + \mathbf{width} \cdot (\mathbf{n} - 1)$ For the simplified *seg* versions of the opcode, the stride ($\mathbf{r_{str}}$) is equal to the segment width ($\mathbf{width} \cdot \mathbf{n}$). For the simplified *st* versions of the opcode, the segment size ($\mathbf{n}$) is 1.

$\mathcal{L} = \{\texttt{lb}, \texttt{lbu}, \texttt{lh}, \texttt{lhu}, \texttt{lw}\}$
$\mathcal{S} = \{\texttt{sb}, \texttt{sh}, \texttt{sw}\}$
$\mathcal{P} = \{(\texttt{c0}/\texttt{sdp}), (!\texttt{c0}/\texttt{sdp})\}$

Table 4.2: Vector load and store commands.

4.2 Virtual Processors

4.2.1 Structure and State

The biggest difference between the Scale architecture and the generic VT architecture in Section 2.5 is the software-visible clustering of resources in Scale's VPs. This change is motivated by the performance and efficiency requirements of embedded applications. Clusters greatly simplify the instruction dependence analysis that the VTU hardware must perform in order to issue multiple instructions per cycle in each lane. They also expose operand locality and enable efficient distributed register files which each have only a few ports. Without clusters, a monolithic multi-ported register file would be needed to issue multiple instructions per cycle.

Each cluster has a configurable number of private and shared registers (pr's and sr's) and a private 1-bit predicate register (p). Additionally, each cluster has two shared chain registers (cr0 and cr1) which correspond to the two input operands for the cluster's functional units. The chain registers can be used to avoid reading and writing the general register file, and they are also implicitly overwritten whenever any instruction executes on the cluster. The memory access cluster (c0) also contains an extra set of private store-data registers (sd's), a store-data predicate register (sdp), and a shared store-data chain register (sdc). Separating the store-data registers simplifies the implementation of vector-store commands and enables store decoupling in the Scale microarchitecture.

4.2.2 VP Instructions

An overview of Scale's VP instructions is shown in Table 4.3. All the clusters in a VP share a common set of capabilities, but they are also individually specialized to support certain extra operations. An instruction executes on a specific cluster and performs a simple RISC-like operation. The source operands for a VP instruction must come from registers local to the cluster or from the cluster's prevVP port, but an instruction may target both local and remote destination registers. Scale allows each VP instruction to have multiple destinations: up to one register in each cluster as well as the local nextVP port. As an example, the following Scale assembly instruction:

```
c0 addu pr0, sr2 -> pr1, c1/sr0, c2/cr0
```

performs an addition on cluster 0 using a private register and a shared register as operands, and it writes the result to a local private register, a remote shared register on cluster 1, and a remote chain register on cluster 2. Most instructions may also use a 5-bit immediate in place of the second operand. The load immediate instruction (li) supports a 32-bit immediate operand, but the assembler/loader may decompose it into multiple primitive machine instructions.

The Scale architecture also provides floating-point instructions for VPs, but since these are not implemented in the Scale prototype they are not listed in Table 4.3. The overall strategy is for cluster 3 to support floating-point multiplies and cluster 2 to support floating-point adds. Similarly, the architecture could be extended with special instructions to support fixed-point saturation and rounding and subword-SIMD operations.

4.2.3 Predication

Predication allows a VP to evaluate small conditionals with no AIB fetch overhead. When a VP instruction targets a predicate register, the low-order bit of the result is used. Any VP instruction can be positively or negatively predicated based on the cluster's local predicate register. For example, the following instruction sequence computes the absolute value of pr0, assuming that sr0 has been set to zero:

Mnemonic	***Operation***
Arithmetic/Logical (all clusters)	
addu	addition
subu	subtraction
and	logical and
or	logical or
xor	logical exclusive or
nor	inverse logical or
sll	shift left logical
srl	shift right logical
sra	shift right arithmetic
seq	set if equal
sne	set if not equal
slt	set if less than
sltu	set if less than unsigned
psel	select based on predicate reg.
copy*	copy
li*	load immediate
la*	load address
Memory (cluster 0)	
lb	load byte (sign-extend)
lbu	load byte unsigned (zero-extend)
lh	load halfword (sign-extend)
lhu	load halfword unsigned (zero-extend)
lw	load word
sb	store byte
sh	store halfword
sw	store word
lw.atomic.add*	atomic load-add-store word
lw.atomic.and*	atomic load-and-store word
lw.atomic.or*	atomic load-or-store word

Mnemonic	***Operation***
Fetch (cluster 1)	
fetch*	fetch AIB at address
psel.fetch	select address based on predicate reg. and fetch AIB
addu.fetch	compute address with addition and fetch AIB
Multiplication/Division (cluster 3)	
mulh	16-bit multiply (signed×signed)
mulhu	16-bit multiply (unsigned×unsigned)
mulhus	16-bit multiply (unsigned×signed)
multu.lo	32-bit multiply (unsigned×unsigned) producing low-order bits
multu.hi	32-bit multiply (unsigned×unsigned) producing high-order bits
mult.lo	32-bit multiply (signed×signed) producing low-order bits
mult.hi	32-bit multiply (signed×signed) producing high-order bits
divu.q	32-bit divide (unsigned/unsigned) producing quotient
div.q	32-bit divide (signed/signed) producing quotient
divu.r	32-bit divide (unsigned÷unsigned) producing remainder
div.r	32-bit divide (signed÷signed) producing remainder

$\star$ Pseudo-instruction, implemented as one or more primitive machine instructions.

Table 4.3: VP instruction opcodes.

```
c0     slt  pr0, sr0 -> p    # p = (pr0 < 0)
c0 (p) subu sr0, pr0 -> pr0  # if (p) pr0 = (0 - pr0)
```

Scale also provides a useful predicate select instruction. For example, the following instruction sequence sets pr2 to the maximum of pr0 and pr1:

```
c0 slt  pr0, pr1 -> p    # p = (pr0 < pr1)
c0 psel pr0, pr1 -> pr2  # pr2 = p ? pr1 : pr0
```

Scale provides only one predicate register per VP, and the logic for computing predicates is performed using regular registers. This is in contrast to some vector architectures which provide multiple mask registers and special mask instructions [Asa98, EML88, Koz02, UIT94, Wat87]. Scale's software interface allows any VP instruction to be optionally predicated, and the single predicate register saves encoding bits by avoiding the need for a register specifier. Also, using regular instructions to compute predicates avoids extra complexity in the hardware. Although computing predicates with regular registers could cause many vector register bits to be wasted in a traditional vector architecture, in Scale the temporaries can be mapped to shared or chain registers. Another concern is that complicated nested conditionals can be difficult to evaluate using a single predicate register [Asa98, SFS00]. However, Scale lets the programmer use AIB fetch instructions instead of predication to map this type of code.

Scale allows any VP instruction to be predicated, including instructions that write destination registers in remote clusters. In the Scale microarchitecture, nullifying remote writes is more difficult than nullifying writes local to a cluster (Section 5.4.3). Nevertheless, we decided that providing cleaner semantics justified the extra hardware complexity.

4.3 Atomic Instruction Blocks

Scale's software-level instruction-set architecture is based on virtual processor instructions that are grouped into atomic instruction blocks. An AIB is the unit of work issued to a VP at one time, and it contains an ordered list of VP instructions with sequential execution semantics. Figure 4-2 shows an example AIB. Although clusters are exposed to software, there are no constraints as to which clusters successive VP instructions can target. In other words, instruction parallelism between clusters is not explicit in Scale's software ISA. This is in contrast to VLIW architectures in which software explicitly schedules operations on parallel functional units. For example, compare the example Scale AIB to the VLIW encoding shown in Figure 4-3.

Scale hardware does not execute software-level VP instructions directly. Instead they are translated into implementation-specific micro-ops, either by the compiler or the loader. The translation is performed at the granularity of AIBs, and the micro-op format for an AIB does expose the parallelism between cluster operations. The micro-op format and translation is described in detail in Section 5.2.1.

Although the inter-cluster instruction ordering is not constrained by the software interface, run-time parallelism within an AIB is generally enhanced by interleaving instructions for different clusters. Interleaving is demonstrated by the example AIB in Figure 4-2. The multiplies for the RGB-to-YCC conversion are performed on cluster 3, the accumulations on cluster 2, and the fixed-point rounding on cluster 1. The example also shows how interleaving can allow chain registers to be used to hold temporary operands between producer and consumer instructions.

A recommended programming pattern when mapping code to virtual processors is for inter-cluster data dependencies within an AIB to have an acyclic structure. This property holds for the AIB in Figure 4-2 since all of the dependencies go from c3 to c2 and from c2 to c1. When such an AIB is vector-fetched, and as a result executed by many VPs in each lane, cluster decoupling in the hardware can allow the execution of different VPs to overlap. For the example AIB, c3, c2, and c1 can all be executing instructions for different VPs at the same time, with c3 running the furthest ahead and c1 trailing behind. This loop-level parallelism can be extracted even if the instructions within an AIB (i.e. within a loop body) are all sequentially dependent with no available parallelism. This is similar to the parallelism exploited by instruction chaining in traditional vector architectures, except that in Scale the instructions are grouped together in an AIB.

4.4 VP Configuration

In the Scale architecture, the control processor configures the VPs by indicating how many shared and private registers are required in each cluster. The length of the virtual processor vector changes with each re-configuration to reflect the maximum number of VPs that can be supported by the hardware. This operation is typically done once outside of each loop, and state in the VPs is undefined across re-configurations.

```
rgbycc_aib:
  .aib begin
  c3 mulh pr0, sr0 -> c2/cr0  # R * Y_R
  c3 mulh pr1, sr1 -> c2/cr1  # G * Y_G
  c2 addu cr0, cr1 -> cr0     # accum. Y
  c3 mulh pr2, sr2 -> c2/cr1  # B * Y_B
  c2 addu cr0, cr1 -> c1/cr0  # accum. Y
  c1 addu cr0, sr0 -> cr0     # round Y
  c1 srl  cr0, 16  -> c0/sd0  # scale Y

  c3 mulh pr0, sr3 -> c2/cr0  # R * CB_R
  c3 mulh pr1, sr4 -> c2/cr1  # G * CB_G
  c2 addu cr0, cr1 -> cr0     # accum. CB
  c3 mulh pr2, sr5 -> c2/cr1  # B * CB_B
  c2 addu cr0, cr1 -> c1/cr0  # accum. CB
  c1 addu cr0, sr1 -> cr0     # round CB
  c1 srl  cr0, 16  -> c0/sd1  # scale CB

  c3 mulh pr0, sr6 -> c2/cr0  # R * CR_R
  c3 mulh pr1, sr7 -> c2/cr1  # G * CR_G
  c2 addu cr0, cr1 -> cr0     # accum. CR
  c3 mulh pr2, sr8 -> c2/cr1  # B * CR_B
  c2 addu cr0, cr1 -> c1/cr0  # accum. CR
  c1 addu cr0, sr1 -> cr0     # round CR
  c1 srl  cr0, 16  -> c0/sd2  # scale CR
  .aib end
```

Figure 4-2: Scale AIB for RGB-to-YCC (based on the generic VT code for RGB-to-YCC shown in Figure 2-21).

C0	C1	C2	C3
			mulh
			mulh
		addu	mulh
		addu	mulh
	addu		mulh
	srl	addu	mulh
		addu	mulh
	addu		mulh
	srl	addu	mulh
		addu	
	addu		
	srl		

Figure 4-3: VLIW encoding for RGB-to-YCC. Only the opcodes are shown. The scheduling assumes single-cycle functional unit latencies.

As an example, consider the configuration command corresponding to the RGB-to-YCC code in Figure 4-2:

```
vcfgvl vlen, n, 3,0, 0,2, 0,0, 3,9
```

This command configures VPs with three private registers in cluster 0 (the store-data registers), two shared registers in cluster 1, no private or shared registers in cluster 2 (since it only uses chain registers), three private registers in cluster 3 to hold the load data, and nine shared registers in cluster 3 to hold the color conversion constants. As a side-effect, the command updates the maximum vector length (vlmax) to reflect the new configuration. In a Scale implementation with 32 physical registers per cluster and four lanes, vlmax will be: $(\lfloor(32-9)/3\rfloor)\times 4 = 28$, limited by the register demands of cluster 3 (refer to Figure 2-20 for a diagram). Additionally, the vcfgvl command sets vl, the active vector length, to the minimum of vlmax and the length argument provided (n), and the resulting length is also written to a destination register (vlen).

Similar to some traditional vector architectures, Scale code can be written to work with any number of VPs, allowing the same binary to run on implementations with varying or configurable resources (i.e. number of lanes or registers). The control processor code simply uses the vector length as a parameter as it stripmines a loop, repeatedly launching a vector's-worth of iterations at a time. Scale's setvl command is used to set the vector length to the minimum of vlmax and the length argument provided (i.e. the number of iterations remaining for the loop). Figure 4-4 shows a simple vector-vector-add example in Scale assembly code.

```
void vvadd(int n, const int32_t* A, const int32_t* B, int32_t* C )
  for (int i=0; i<n; i++) {
    C[i] = A[i] + B[i];
  }
}
```

```
vvadd:
  # configure VPs: c0:p,s c1:p,s c2:p,s c3:p,s
  vcfgvl  vlen, n,    1,0,   2,0,   0,0,   0,0
                                       #   vlmax = [ impl. dependent ]
                                       #   (vl, vlen) = MIN(n, vlmax)
  sll     stride, vlen, 2              # stride = vlmax*4

stripmine_loop:
  setvl   vlen, n                 # set vector length:
                                  #   (vl, vlen) = MIN(n, vlmax)
  vlwai   srcA, stride, c1/pr0    # vector-load A input
  vlwai   srcB, stride, c1/pr1    # vector-load B input
  vf      vvadd_aib               # vector-fetch vvadd_aib
  vswai   dstC, stride, c0/sd0    # vector-store C output
  subu    n, vlen                 # decrement length by vlen
  bnez    n, stripmine_loop       # branch to stripmine loop
  vsync                           # wait for VPs to finish
  jr      ra                      # function call return

vvadd_aib:
  .aib begin
  c1 addu pr0, pr1 -> c0/sd0      # do add
  .aib end
```

Figure 4-4: Vector-vector add example in C and Scale assembly code.

4.5 VP Threads

As in the generic VT architecture, Scale's virtual processors execute threads comprising a sequence of atomic instruction blocks. An AIB can have several predicated fetch instructions, but at most one fetch may issue in each AIB. If an AIB issues a fetch, the VP thread persists and executes the target AIB, otherwise the VP thread stops. A fetch instruction may be anywhere within an AIB sequence (not just at the end), but the entire current AIB always executes to completion before starting the newly fetched AIB.

Scale provides cluster 1 with fetch instructions where the target AIB address is either calculated with an addition (`addu.fetch`) or chosen with a predicate select (`psel.fetch`). The AIB addresses used for VP fetches may be written to shared registers by the control processor. This is commonly done when a loop body contains a simple conditional block or an inner loop. For example, this sequence can implement an if-then-else conditional:

```
c2 slt pr0, pr1 -> c1/p  # if (pr0 < pr1)
c1 psel.fetch sr0, sr1   # fetch sr1, else fetch sr0
```

And this sequence can implement an inner loop executed by the VP thread:

```
c1      subu  pr0, 1 -> pr0  # decrement loop counter
c1      seq   pr0, 0 -> p    # compare count to 0
c1 (!p) fetch sr0            # fetch loop unless done
```

```
aibA:
  .aib begin
  [...]
  c1  (p) addu.fetch pr0, (aibB - aibA) -> pr0  # fetch aibB if (p)
  c1 (!p) addu.fetch pr0, (aibC - aibA) -> pr0  # fetch aibC if (!p)
  [...]
  .aib end

aibB:
  .aib begin
  [...]
  c1 addu.fetch pr0, (aibD - aibB) -> pr0  # fetch aibD
  [...]
  .aib end

aibC:
  .aib begin
  [...]
  c1 addu.fetch pr0, (aibD - aibC) -> pr0  # fetch aibD
  [...]
  .aib end
```

Figure 4-5: VP thread program counter example. By convention, the current AIB address is held in c1/pr0. Each fetch instruction produces the new AIB address as a result and writes it to this register. The immediate arguments for the `addu.fetch` instructions are expressed as the difference between AIB labels. The linker calculates these immediates at compile time.

VP threads may also have complex control flow and execute long sequences of AIBs. To manage the AIB addresses in this case, a VP will usually maintain its own “program counter” in a private register. By convention, this register always holds the address of the current AIB, and it is updated whenever a new AIB is fetched. An example of this style of VP code is shown in Figure 4-5.

The VP threads operate concurrently in the same address space. When they read and write shared memory locations, they must use atomic memory instructions (Table 4.3) to ensure coherence. Higher-level synchronization and mutual exclusion primitives can be built on top of these basic instructions.

4.6 Cross-VP Communication

Each cluster in a virtual processor connects to its siblings in neighboring VPs with a prevVP input port that can be read as a source operand and a nextVP output port that can be written as a destination (refer to Figure 4-1). Typically, cross-VP chains are introduced by vector-fetching an AIB which contains prevVP receives and nextVP sends. VP0 receives prevVP input data from the per-cluster cross-VP start/stop queues, and—before vector-fetching the first cross-VP chain—the control processor uses xvppush commands to write the initial values to these queues. The cross-VP network has a ring structure, meaning that nextVP data from the last VP wraps back around to the cross-VP start/stop queue. The data can then by used by the next vector-fetched cross-VP chain, or the control processor can consume it with an xvppop or xvpdrop command.

Cross-VP data transfers allow a VT architecture to execute loops with cross-iteration dependencies (also known as loop-carried dependencies). As an example, Figure 4-6 shows a simplified version of the ADPCM speech decoder algorithm. The example loop has two cross-iteration dependencies (index and valpred), and these are evaluated separately by c1 and c2. The control thread

```
void decode_ex(int len, u_int8_t* in, int16_t* out) {
  int i;
  int index = 0;
  int valpred = 0;
  for(i = 0; i < len; i++) {
    u_int8_t delta = in[i];
    index += indexTable[delta];
    index = index < IX_MIN ? IX_MIN : index;
    index = IX_MAX < index ? IX_MAX : index;
    valpred += stepsizeTable[index] * delta;
    valpred = valpred < VALP_MIN ? VALP_MIN : valpred;
    valpred = VALP_MAX < valpred ? VALP_MAX : valpred;
    out[i] = valpred;
  }
}
```

```
decode_ex: # a0=len, a1=in, a2=out
  # configure VPs: c0:p,s c1:p,s c2:p,s c3:p,s
  vcfgvl  t1, a0,     1,2,   0,3,   0,3,   0,0
  # (vl,t1) = min(a0,vlmax)
  sll     t1, t1, 1       # output stride
  la      t0, indexTable
  vwrsh   t0, c0/sr0      # indexTable addr.
  la      t0, stepsizeTable
  vwrsh   t0, c0/sr1      # stepsizeTable addr.
  vwrsh   IX_MIN, c1/sr0  # index min
  vwrsh   IX_MAX, c1/sr1  # index max
  vwrsh   VALP_MIN, c2/sr0# valpred min
  vwrsh   VALP_MAX, c2/sr1# valpred max
  xvppush $0, c1          # push initial index = 0
  xvppush $0, c2          # push initial valpred = 0
stripmineloop:
  setvl   t2, a0          # (vl,t2) = min(a0,vlmax)
  vlbuai  a1, t2, c0/pr0  # vector-load input, incr. ptr
  vf      vtu_decode_ex   # vector-fetch AIB
  vshai   a2, t1, c0/sd0  # vector-store output, incr. ptr
  subu    a0, t2          # decrement count
  bnez    a0, stripmineloop # loop until done
  xvpdrop c1              # pop final index, discard
  xvpdrop c2              # pop final valpred, discard
  vsync                   # wait until VPs are done
  jr ra                   # return

vtu_decode_ex:
  .aib begin
  c0 sll  pr0, 2      -> cr1            # word offset
  c0 lw   cr1(sr0)    -> c1/cr0         # load index
  c0 copy pr0         -> c3/cr0         # copy delta
  c1 addu cr0, prevVP -> cr0            # accumulate index    ( prevVP recv )
  c1 slt  cr0, sr0    -> p              # index min
  c1 psel cr0, sr0    -> sr2            # index min
  c1 slt  sr1, sr2    -> p              # index max
  c1 psel sr2, sr1    -> c0/cr0, nextVP # index max           ( nextVP send )
  c0 sll  cr0, 2      -> cr1            # word offset
  c0 lw   cr1(sr1)    -> c3/cr1         # load step
  c3 mulh cr0, cr1    -> c2/cr0         # step*delta
  c2 addu cr0, prevVP -> cr0            # accumulate valpred  ( prevVP recv )
  c2 slt  cr0, sr0    -> p              # valpred min
  c2 psel cr0, sr0    -> sr2            # valpred min
  c2 slt  sr1, sr2    -> p              # valpred max
  c2 psel sr2, sr1    -> c0/sd0, nextVP # valpred max         ( nextVP send )
  .aib end
```

Figure 4-6: Decoder example in C code and Scale code.

provides the initial values with xvppush commands before entering the stripmined loop. Within the stripmined loop, vector-fetches repeatedly launch the cross-VP chains, and vector-loads and vector-stores read and write memory. Once all the iterations are complete, the final values are discarded with xvpdrop commands.

The prevVP receives and nextVP sends within a vector-fetched AIB should be arranged in corresponding pairs. That is to say, when a virtual processor receives a cross-VP data value on its prevVP input port, it should send out a corresponding cross-VP data value on its nextVP output port. Even if instructions which read prevVP or write nextVP are predicated, one nextVP send should issue for each prevVP receive that issues. An AIB may read/write multiple values from/to the prevVP and nextVP ports, in which case the order of the receives and sends should correspond; i.e. communication through the ports is first-in-first-out.

4.6.1 Queue Size and Deadlock

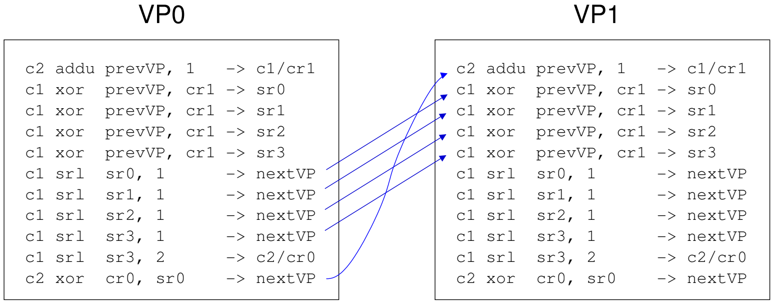
Whenever queues are explicit in a hardware/software interface, the size must be exposed in some way so that software can avoid overflow or deadlock. In Scale, the per-cluster cross-VP start/stop queues can each hold 4 data elements (up to 16 values across the clusters). This is also the maximum number of prevVP/nextVP cross-VP data transfers that are allowed in one AIB.

When a vector-fetched AIB creates a chain of cross-VP sends and receives—and the initial data has been written to the cross-VP start/stop queue—deadlock will be avoided as long as the VPs are able to execute their AIBs in order (VP0 to VPN) without blocking. When a virtual processor encounters a prevVP receive, it will stall until the data is available. When a VP executes a nextVP send, either the receiver must be ready to receive the data, or the data must be buffered. To guarantee that deadlock will not occur, the Scale architecture dictates that a later VP can never cause an earlier VP to block. As a result, Scale must provide buffering for the maximum amount of cross-VP data that an AIB can generate. This buffering is provided in the microarchitecture by the cross-VP queues described in Section 5.4.1. With this guarantee, the VPs can each—in order, starting with VP0—execute the entire AIB, receiving all cross-VP data from their predecessor and producing all cross-VP data for their successor. As a result, a cross-VP chain created by a vector-fetched AIB will evaluate with no possible deadlock.

It is tempting to try to add restrictions to the software interface to eliminate the need for cross-VP data buffering in the VT hardware. For example, in a VT machine with one cluster, if all prevVP receives are required to precede all nextVP sends within an AIB, then an implementation does not need to provide buffering between VPs. As long as the VPs are executing concurrently in the hardware, a later VP can always receive all the cross-VP data from its predecessor, allowing the predecessor to execute the AIB without blocking. If there are two or more lanes in the hardware, then the VPs can each execute the entire AIB in succession to complete a vector-fetched cross-VP chain with no possible deadlock.

Although buffering can be eliminated if there is only one cluster, multiple clusters can have subtle interactions with cross-VP dependencies. Consider the somewhat contrived, but simple AIB in Figure 4-7. In this example, a VP must execute all of the nextVP instructions on cluster 1 before it executes the nextVP instruction on cluster 2. However, the successor VP must receive the prevVP data for cluster 2 before it will be ready to receive any of the prevVP data for cluster 1. Thus, an implementation must provide buffering for the nextVP data on cluster 1 in order to avoid deadlock. One way around this possibly surprising result would be to restrict the nextVP sends to execute in the same order as the prevVP receives; this would eliminate the criss-crossing dependency arrows in Figure 4-7.

As demonstrated by this example, eliminating the need for cross-VP data buffering would com-



```
c2 addu prevVP, 1   -> c1/cr1
c1 xor  prevVP, cr1 -> sr0
c1 xor  prevVP, cr1 -> sr1
c1 xor  prevVP, cr1 -> sr2
c1 xor  prevVP, cr1 -> sr3
c1 srl  sr0, 1      -> nextVP
c1 srl  sr1, 1      -> nextVP
c1 srl  sr2, 1      -> nextVP
c1 srl  sr3, 1      -> nextVP
c1 srl  sr3, 2      -> c2/cr0
c2 xor  cr0, sr0    -> nextVP
```

```
c2 addu prevVP, 1   -> c1/cr1
c1 xor  prevVP, cr1 -> sr0
c1 xor  prevVP, cr1 -> sr1
c1 xor  prevVP, cr1 -> sr2
c1 xor  prevVP, cr1 -> sr3
c1 srl  sr0, 1      -> nextVP
c1 srl  sr1, 1      -> nextVP
c1 srl  sr2, 1      -> nextVP
c1 srl  sr3, 1      -> nextVP
c1 srl  sr3, 2      -> c2/cr0
c2 xor  cr0, sr0    -> nextVP
```

Figure 4-7: Example showing the need for cross-VP buffering. The AIB is shown for VP0 and VP1, and the cross-VP dependencies are highlighted.

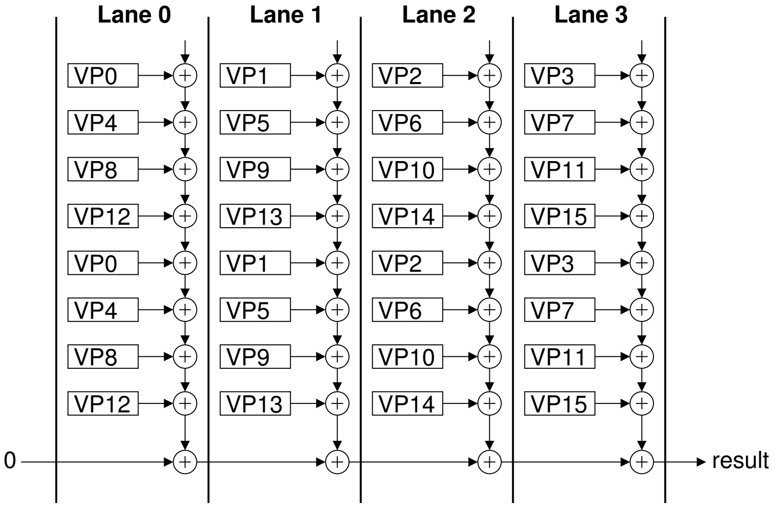


Figure 4-8: Reduction example. The diagram shows a sum reduction accumulated first within 4 lanes, and then the final result accumulated across the 4 lanes.

plicate the software interface. Requiring all receives to precede all sends could also hurt performance. Therefore, when designing Scale, we decided that providing a small amount of cross-VP buffering in each lane was an acceptable trade-off.

4.7 Independent VPs

Scale extends the baseline VT architecture with the concept of *independent VPs* that execute in parallel rather than being time-multiplexed. Independent VPs are used for two primary purposes: (1) to compute associative reductions, and (2) to allow arbitrary cross-VP communication between VPs. Scale's software interface defines a *vector length for independent VPs* (`vlind`), and in the Scale implementation `vlind` is equal to the number of lanes.

4.7.1 Reductions

The shared registers in a VT architecture provide an efficient way to compute associative reduction operations, e.g. calculating the sum, maximum, or minimum of a vector. The VPs first use a shared register to accumulate a partial result in each lane. This is done using a regular stripmined loop with a long vector length (and hence many VPs mapped to each lane). Then, the vector length is set to `vlind` to accumulate the final result across the lanes using the cross-VP network. A diagram of such an operation is shown in Figure 4-8.

4.7.2 Arbitrary Cross-VP Communication

As described above, the standard way for software to use the cross-VP network is for the control processor to vector-fetch an AIB that contains corresponding prevVP receives and nextVP sends. This establishes a chain of cross-VP dependencies, and it guarantees that each send has a corresponding receive.

One could also imagine using the cross-VP network in a more general context, as a communication link between VP threads that have internal control flow. In this case, the nextVP send and its corresponding prevVP receive are not necessarily in the same AIB. Instead, they come from arbitrary AIBs that the sending and receiving VPs have thread-fetched. Although it seems like an intuitive usage pattern, the Scale architecture does not allow this type of arbitrary cross-VP communication between VP threads. This restriction enables an efficient hardware implementation—the virtual processors are time-multiplexed on the physical lanes in the VTU, and the cross-VP links between lanes are shared by the VPs. With this structure, if arbitrary communication between VP threads were allowed, a nextVP send from any VP thread mapped to lane 0 could inadvertently pair with a prevVP receive from any VP thread mapped to lane 1. Compared to vector-fetched AIBs which have a well-defined execution order on the VTU, the VPs may be arbitrarily interleaved when executing thread-fetched AIBs.

Although Scale restricts cross-VP communication in the general case, when the vector length is set to `vlind` VP threads with arbitrary control flow are allowed to communicate freely using the cross-VP network. Since the hardware resources are not time-multiplexed, the cross-VP sends and receives are always paired correctly.

Independent VPs allow loops with cross-iteration dependencies and arbitrary control flow to be mapped to Scale, but the mechanism does limit the vector length and thereby the parallelism that is available for the hardware to exploit. Nevertheless, this is often not a major limitation since the cross-iteration dependencies themselves are likely to limit the available parallelism in these types of loops anyways. In fact, the cross-VP network is most beneficial when mapping loops with simple cross-iteration dependencies whose latency can be hidden by the parallel execution of a modest number of other instructions within the loop body. These loops can usually fit within a single vector-fetched AIB, so the Scale architecture is optimized for this case.

4.8 Vector Memory Access

Scale's vector memory commands (Table 4.2) expose structured memory accesses patterns, allowing an implementation to efficiently drive a high bandwidth memory system while hiding long latencies. Like vector-fetch commands, these operate across the virtual processor vector; a vector-load writes the load data to a private register in each VP, while a vector-store reads the store data from a private store-data register in each VP. Scale supports unit-stride and segment-strided vector memory access patterns (Section 2.7.4). The vector-store commands may be predicated based on cluster 0's private store-data predicate register (sdp). Scale also supports vector-load commands which target shared registers to retrieve values used by all VPs. Although not shown in Table 4.2, Scale's shared vector-loads may also retrieve multiple elements, optionally with a stride between elements.

Vector load and store commands are generally interleaved with vector-fetch commands which process the load data and generate the store data. The Scale implementation is able to overlap the execution of vector load, store, and fetch commands via chaining. For this reason, it is often beneficial to partition a vectorizable loop into multiple AIBs whose execution can overlap with the different vector memory commands. For example, instead of performing all of the vector-loads before issuing a vector-fetch, the control processor can vector-fetch an AIB after each vector-load to perform a slice of the computation. Partitioning AIBs in this way can increase the private register demands when values need to be communicated from one AIB to another. However, in other cases the partitioning can reduce the private register count by allowing the vector-loads and vector-stores to reuse registers.

4.9 Control Processor and VPs: Interaction and Interleaving

Scale's program semantics are based on those defined for the generic VT architecture in Section 2.6. The Scale virtual machine (Scale VM) is essentially a functional version of the programmer's model in Figure 4-1. It includes a control processor and a configurable vector of virtual processors that execute blocks of VP instructions. These processors operate concurrently in a shared memory address space.

The control processor uses a vector-fetch command to send an AIB to the virtual processor vector, and—if the AIB itself contains fetch instructions—to launch a set of VP threads. Additionally the control processor can use a VP-fetch command to launch an individual VP thread.

As in the generic VT architecture, once it is launched a VP thread will execute to completion before processing a subsequent regular control processor command. For example, a VP register-read or register-write command following a vector-fetch will not happen until after the VP thread completes. However, Scale also provides special non-blocking versions of the VP register-read, register-write, and fetch commands (Table 4.1). The non-blocking commands issue asynchronously with respect to the execution of the VP thread. They allow the control processor to monitor or communicate with a running VP thread, and they can also aid in debugging.

To resolve memory ordering dependencies between VPs (and between itself and the VPs), the control processor can use a `vsync` command to ensure that all previous commands have completed before it proceeds. It can also use a `vfence` command to ensure that all previous commands complete before any subsequent commands execute—in this case, the CP itself does not have to stall. Note that these commands are used to synchronize vector-loads and vector-stores as well as the VP threads.

4.10 Exceptions and Interrupts

The Scale control processor handles exceptions and interrupts. When such an event occurs, the processor automatically jumps to a pre-defined kernel address in a privileged execution mode. Depending on the exceptional condition, the handler code can optionally use the `vkill` command to reset the vector-thread unit into an idle state.

Exceptions within the VTU are not precise, and a `vkill` aborts previously issued VTU commands at an arbitrary point of partial execution. However, the Scale architecture can still support restartable exceptions like virtual memory page faults. This is achieved at the software level by dividing the code into idempotent regions and using software restart markers for recovery [HA06]. With minor additions to the Scale instruction set, a similar software technique could make individual VP threads restartable. A more thorough description of restart markers and exception handling is beyond the scope of this thesis.

Incorrect user code can cause deadlock in the vector-thread unit, for example by using the cross-VP network improperly. An incorrect translation from VP instructions to hardware micro-ops could also cause the VTU to deadlock. Other problems in user code, like infinite loops in VP threads, can lead to livelock in the vector-thread unit. The deadlock conditions can potentially be detected by watchdog progress monitors in the hardware, while livelock detection will generally require user intervention (e.g. using control-C to terminate the program). In either case, an interrupt is generated for the control processor, and system software can recover from the error by using the `vkill` command to abort the vector-thread unit execution.

4.11 Scale Virtual Machine Binary Encoding and Simulator

In addition to serving as a conceptual model, Scale VM is used to define a binary encoding for Scale programs, and it is implemented as an instruction set simulator. The Scale VM simulator provides a high-level functional execution of Scale VM binaries, and it serves as a golden model for defining the reference behavior of Scale programs.

4.11.1 Scale Primop Interpreter

A virtual processor in Scale VM is defined in terms of a Scale Primop Interpreter (SPI, pronounced “spy”). A SPI is a flexible software engine that executes a set of primitive operations (primops). Each Scale VP instruction is decomposed into several primops. This indirection through primops facilitates architectural exploration, as different Scale instruction sets can be evaluated without changing the underlying primop interpreter.

A SPI is similar to an abstract RISC processor and it includes a set of foreground registers. Primops use a fixed-width 32-bit encoding format, and a basic primop includes two source register specifiers, a destination register specifier, and an opcode. Primops include basic arithmetic and logic operations, loads and stores, and a set of bit manipulation operations that are useful for instruction set interpretation. A SPI also includes a large number of background registers, and additional primops can copy registers between the foreground and background. SPIs also have input and output queue ports. These ports enable direct communication between different SPIs when they are linked together externally. Push and pop primops allow a SPI to write and read the queue ports. A SPI blocks when it tries to pop from an empty queue.

A set of conventions define the mapping of Scale virtual processors onto Scale primop interpreters. A set of background registers are used to hold each cluster’s register state. Likewise, the prevVP and nextVP ports for the different clusters map to certain SPI queue ports. As an example mapping, the following Scale VP instruction:

```
c2 addu cr0, pr0 -> c1/cr0, c0/pr1
```

Is implemented by the following sequence of primops:

```
fg $r1, $B2/$b32    # bring c2/cr0 to foreground
fg $r2, $B2/$b0     # bring c2/pr0 to foreground
add $r0, $r1, $r2   # do addition
bg $B1/$b32, $r0    # write result to c1/cr0
bg $B0/$b1, $r0     # write result to c0/pr1
```

Analogous to the execution of atomic instruction blocks by Scale virtual processors, SPIs execute blocks of primops. A SPI has two fetch queues which hold addresses of fetch-blocks to be executed. A controller outside of the SPI may enqueue a fetch-block address in its external fetch queue, and the SPI itself can execute a fetch primop which enqueues a fetch-block address in its internal fetch queue. A SPI always processes fetches from its internal queue before processing a fetch in its external queue. To process a fetch, the SPI sequentially executes all of the primops in the target fetch-block.

4.11.2 Scale VM Binary Encoding

Scale assembly code is composed of two different instruction sets, the RISC instructions for the control processor and the Scale VP instructions for the vector-thread unit. During compilation, the control processor instructions—including the vector-thread unit commands—are translated by the assembler into binary machine instructions. This is a straightforward conversion that is usually a one-to-one mapping from assembly instruction to machine instruction.

Scale VP instructions may appear in the same assembly files as the control processor instructions, and both types of code are located within the `.text` section of the program. VP instructions are demarcated by `.aib begin` and `.aib end` assembler directives. During compilation, the VP instructions are first translated into primop assembly code by the Scale ISA translator (sisa-translator). The assembler then converts the primops into 32-bit machine instructions within the program binary. To help other tools process Scale binary files, the assembler creates a special section which enumerates the addresses of all the AIBs.

The primops in the Scale VM binary format effectively provide a flexible variable-width encoding for Scale VP instructions. These primops are not directly executed by Scale hardware. Instead, Scale VP instructions are translated into a hardware micro-op binary format as described in Section 5.2.1.

4.11.3 Scale VM Simulator

The Scale VM simulator includes a control processor model and an array of SPIs configured to model the virtual processor vector. The main execution loop in the simulator alternates between running the control processor and running the SPIs. When the control processor executes a vector-fetch command, the fetch-block address is queued in the external fetch queue of the SPIs corresponding to the active VPs. A SPI may temporarily block if it tries to executes a pop primop and the data is not available on its input queue port, but the simulator will repeatedly execute the SPIs until they all run to completion.

To aid in program debugging, the Scale VM simulator tracks the virtual processor configuration and detects when programs use invalid registers. The simulator also provides tracing capabilities and it supports special pseudo primops to print debugging information in-line during execution.

4.11.4 Simulating Nondeterminism

To help explore different possible executions of Scale programs, the Scale VM simulator supports both eager and lazy execution modes for the SPIs. Under eager execution, the SPIs run as far as possible after each command from the control processor. In the lazy mode, execution is deferred until a command from the control processor forces a synchronization. For example, many vector-fetch commands may issue and enqueue fetches without actually running the SPIs, but to preserve register dependencies the simulator must run the SPIs to completion before a subsequent vector-load or vector-store can execute.

In addition to lazy and eager execution, the Scale VM simulator also supports different execution interleavings of the SPIs themselves. The simulator always iterates through the SPI array and runs one at a time. The SPIs may either be configured to each run continuously for as long as possible, to each execute one fetch-block at a time, or to each execute one primop at a time.

Nondeterminism manifests as memory races between loads and stores in different threads. To help with program verification and debugging, the Scale VM simulator includes a memory race detector (MRD). The MRD logs all the loads and stores made by different threads. Then, at synchronization points, the MRD compares addresses between different threads to determine if a possible memory race has occurred. A few limitations of the current MRD in Scale VM are: (1) it does not take atomic memory operations into account, so it can falsely detect races between properly synchronized threads; (2) it does not track data values, so two stores may be flagged as a race when they actually store the same value; and, (3) it does not take register dependencies into account, so it may falsely detect races that are precluded by true data dependencies. In explanation of this last point, when a vector-store follows a vector-load and their memory footprints overlap, a synchronizing fence is needed to avoid memory races—*except* for the special case in which the vector-load and the vector-store read and write the exact same addresses and they are ordered by data dependencies through registers. This special case is actually fairly common, it frequently occurs when a vector-load/compute/vector-store pattern is used to modify a data structure in place. Nevertheless, despite these false-positives, the MRD in the Scale VM simulator has been useful in catching potential race conditions.

Chapter 5

Scale VT Microarchitecture

The Scale vector-thread architecture provides a programmer's model based on virtual processors. This abstraction allows software to flexibly and compactly encode large amounts of structured parallelism. However, the level of indirection means that the hardware must dynamically manage and schedule the execution of these VPs. The Scale microarchitecture includes many novel mechanisms to implement vector-threading efficiently. It exploits the VT encoding to execute operations in parallel while using locality to amortize overheads.

This chapter provides a detailed description of the organization and design of the Scale vector-thread microarchitecture. The general structure of the Scale microarchitecture could apply to many possible implementations, but to aid understanding this chapter describes the actual bit-level encodings and parameters (e.g. queue sizes) used in the Scale VT processor prototype.

5.1 Overview

Scale's microarchitecture reflects the structured parallelism and locality exposed by the VT software interface. Scale extends the VT archetype from Section 2.5, with the primary difference being the introduction of multi-cluster lanes. The partitioning of execution resources into clusters provides high computational throughput while limiting the complexity of control and datapaths. Scale also exploits the compact encoding of VT parallelism to improve performance-efficiency through extensive hardware decoupling. Each microarchitectural unit is an independent engine with a simple in-order execution flow, but decoupling allows some units to run far ahead of others in the overall program execution. Decoupling is enabled by simple queues which buffer data and control information for the trailing units to allow the leading units to run ahead.

Figure 5-1 shows an overview of the Scale microarchitecture. The single-issue RISC control processor executes the program control code and issues commands to the vector-thread unit. These include VTU configuration commands, vector-fetches, vector-loads, and vector-stores. Commands are queued in the VTU command queue, allowing the control processor to run ahead.

The VTU includes a parallel array of lanes which are each partitioned into a set of execution clusters. Each cluster contains a register file and an ALU and acts as an independent processing engine with its own small AIB cache. The clusters in a lane are interconnected with a data network, and they communicate with each other using decoupled transport and writeback pipelines. The clusters are also interconnected with a ring network across the lanes to implement cross-VP communication. Each lane also contains a command management unit which processes commands from the VTU and handles thread fetches generated by the VPs mapped to the lane. The CMU maintains tags for the cluster AIB caches and it coordinates with the AIB fill unit to manage a refill.

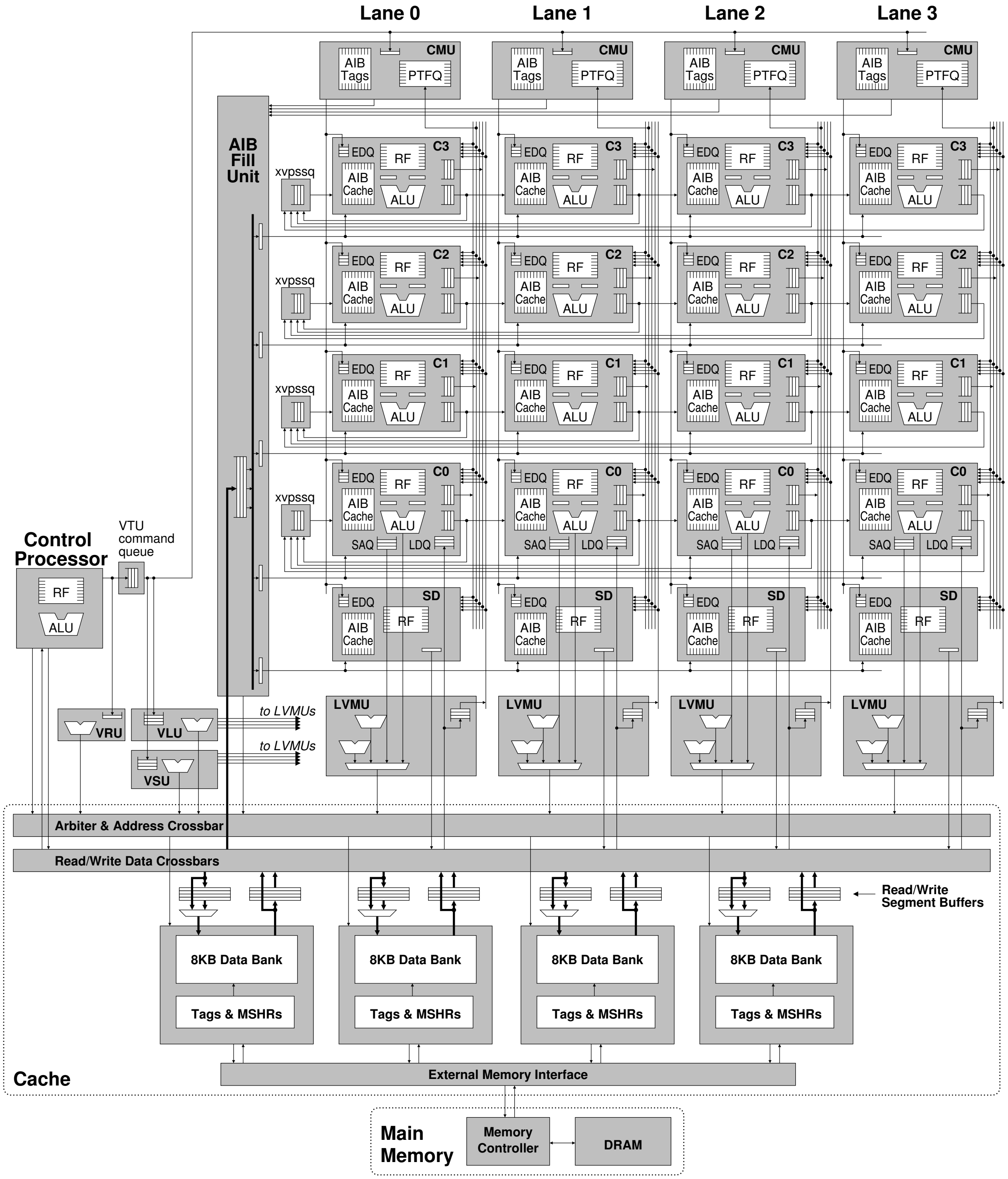


Figure 5-1: Scale microarchitecture overview.

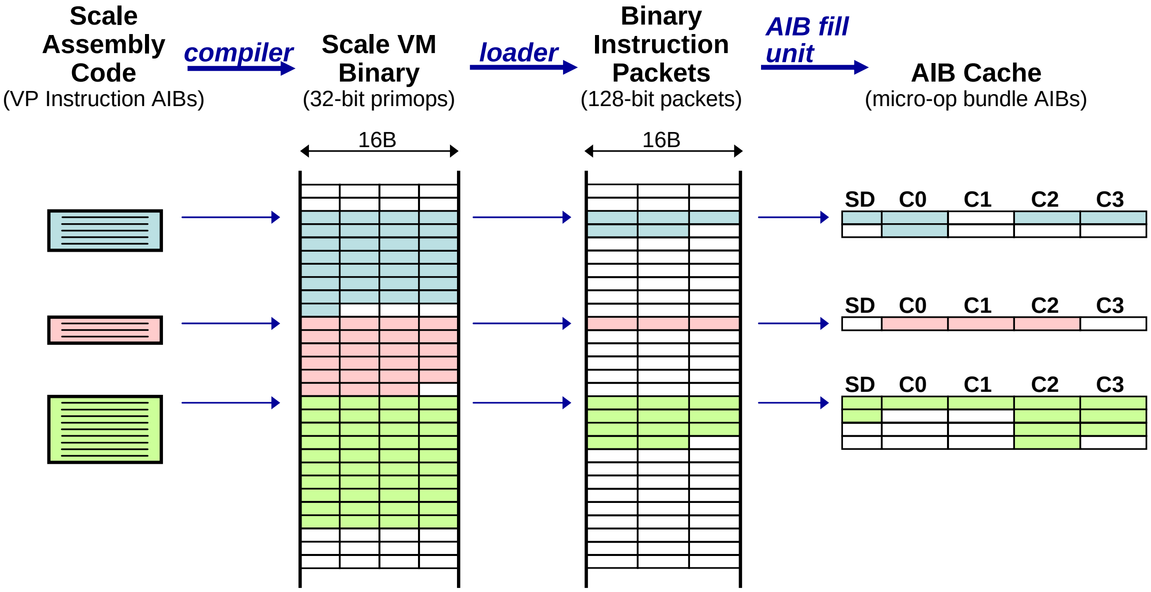


Figure 5-2: Instruction organization overview. The compiler translates Scale assembly code (VP instructions) into the Scale VM binary format (primops). The loader translates the Scale VM encoding into binary instruction packets which are stored in the program memory. During execution, the AIB fill unit converts the binary instruction packets into per-cluster micro-op bundles which are written to the cluster AIB caches.

The CMU distributes work to the clusters in the form of compact execute directives.

The vector load unit (VLU) and the vector store unit (VSU) process vector memory commands from the control processor and manage the interface between the VTU lanes and the memory system. They each include an address generator for unit-stride accesses, and per-lane address generators for segment-strided accesses. The per-lane portions of the vector load and store units are organized as lane vector memory units (LVMUs). The vector refill unit (VRU) preprocesses vector-load commands to bring data into the cache.

Scale's memory system includes a 32 KB first-level unified instruction and data cache, backed up by a large main memory. The cache is comprised of four independent banks, and an arbiter decides which requester can access each bank every cycle. The cache connects to the VTU and the control processor with read and write crossbars, and it also contains load and store segment buffers. The cache is non-blocking and includes miss status handling registers (MSHRs) to continue satisfying new requests even when there are many outstanding misses. As a result, the memory system may return load data out-of-order, and requesters must tag each outstanding access with a unique identifier.

The remainder of this chapter describes these units and their interaction in much greater detail.

5.2 Instruction Organization

The Scale software ISA is portable across multiple Scale implementations but is designed to be easy to translate into implementation-specific micro-operations, or *micro-ops*. The loader translates the Scale software ISA into the native hardware ISA at program load time. It processes a Scale VM binary (described in Section 4.11.2) and essentially converts the primops back into a representation of the original Scale VP instructions. The loader then translates the Scale VP instructions into the

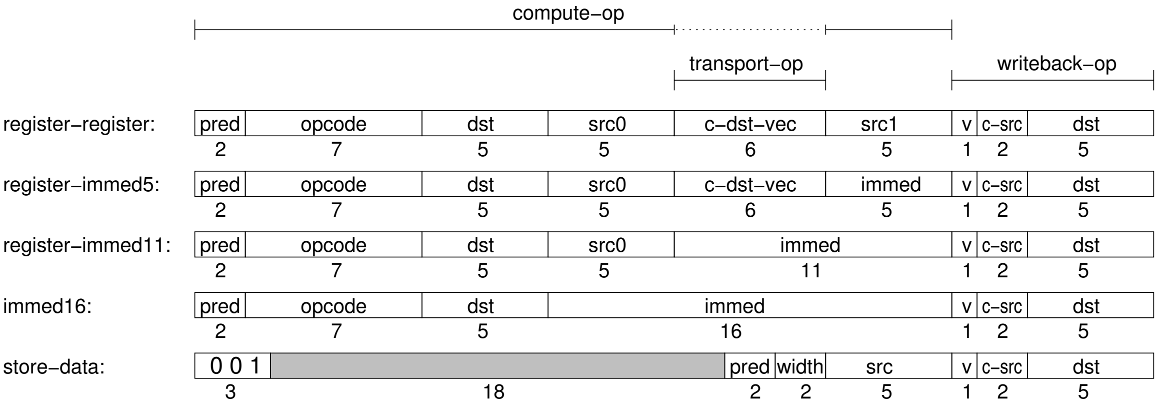

compute–op
transport–op
writeback–op
register–register:
pred 2 | opcode 7 | dst 5 | src0 5 | c–dst–vec 6 | src1 5 | v 1 | c–src 2 | dst 5
register–immed5:
pred 2 | opcode 7 | dst 5 | src0 5 | c–dst–vec 6 | immed 5 | v 1 | c–src 2 | dst 5
register–immed11:
pred 2 | opcode 7 | dst 5 | src0 5 | immed 11 | v 1 | c–src 2 | dst 5
immed16:
pred 2 | opcode 7 | dst 5 | immed 16 | v 1 | c–src 2 | dst 5
store–data:
0 0 1 3 | 18 | pred 2 | width 2 | src 5 | v 1 | c–src 2 | dst 5

Figure 5-3: Micro-op bundle encoding. Each micro-op bundle is 38-bits wide. Regular register-register and register-immed5 micro-op bundles include a compute-op, a transport-op, and a writeback-op. The register-immed11 and immed16 encodings provide longer immediates in place of the transport-op.

For each AIB:

1. Start with an empty sequence of micro-ops for each cluster.
2. For each VP instruction:
   (a) Convert the main operation into a compute-op, possibly with a local destination. Attach an empty transport-op to the compute-op. Append the compute-op/transport-op pair to the associated cluster's micro-op bundle sequence, merging with a previous writeback-op if one is present in the tail micro-op bundle.
   (b) For each remote destination, generate a writeback-op and set the associated bit in the transport-op. Append the writeback-op as a new micro-op bundle in the target cluster's micro-op bundle sequence.

Figure 5-4: Algorithm for converting AIBs from VP instructions to micro-ops.

hardware binary format, one AIB at a time. The binary format compresses micro-ops into 128-bit *binary instruction packets*. To process an AIB cache miss, the AIB fill unit reads these instruction packets from memory and constructs lane-wide fill packets which it broadcasts to the cluster AIB caches. Figure 5-2 shows an overview of the instruction organization.

5.2.1 Micro-Ops

The primary purpose of micro-ops is to handle inter-cluster communication. Scale VP instructions are decomposed into three categories of micro-ops: a *compute-op* performs the main RISC-like operation, a *transport-op* sends data to another cluster, and, a *writeback-op* receives data sent from an external cluster. The loader organizes micro-ops derived from an AIB into *micro-op bundles* which target a single cluster and do not access other clusters' registers. Figure 5-3 shows the encoding of a 38-bit Scale micro-op bundle, and Table 5.1 shows the encoding of the opcode field.

A straightforward algorithm shown in Figure 5-4 converts Scale VP instructions into micro-

opcode		*bits 5..6*			
		0	1	2	3
bits 0..4		00	01	10	11
0	00000	or.rr	or.rr.p	or.ri	or.ri.p
1	00001	nor.rr	nor.rr.p	nor.ri	nor.ri.p
2	00010	and.rr	and.rr.p	and.ri	and.ri.p
3	00011	xor.rr	xor.rr.p	xor.ri	xor.ri.p
4	00100	seq.rr	seq.rr.p	seq.ri	seq.ri.p
5	00101	sne.rr	sne.rr.p	sne.ri	sne.ri.p
6	00110	slt.rr	slt.rr.p	slt.ri	slt.ri.p
7	00111	sltu.rr	sltu.rr.p	sltu.ri	sltu.ri.p
8	01000	srl.rr	srl.rr.p	srl.ri	srl.ri.p
9	01001	sra.rr		sra.ri	
10	01010	sll.rr		sll.ri	
11	01011	addu.rr		addu.ri	addu.ri11.f^{c1}
12	01100	subu.rr		subu.ri	
13	01101	psel.rr	psel.rr.f^{c1}	psel.ri	
14	01110				
15	01111				

opcode		*bits 5..6*			
		0	1	2	3
bits 0..4		00	01	10	11
16	10000	lb.rr^{c0}		lb.ri^{c0}	
17	10001	lbu.rr^{c0}		lbu.ri^{c0}	
18	10010	lh.rr^{c0}		lh.ri^{c0}	
19	10011	lhu.rr^{c0}		lhu.ri^{c0}	
20	10100	lw.rr^{c0}	lw.rr.lock^{c0}	lw.ri^{c0}	lw.ri.lock^{c0}
21	10101	sb.rr^{c0}		sb.ri^{c0}	
22	10110	sh.rr^{c0}		sh.ri^{c0}	
23	10111	sw.rr^{c0}		sw.ri^{c0}	
24	11000	mulh.rr^{c3}	mulhu.rr^{c3}	mulh.ri^{c3}	mulhu.ri^{c3}
25	11001	mulhus.rr^{c3}		mulhus.ri^{c3}	
26	11010	mult.lo.rr^{c3}	multu.lo.rr^{c3}	mult.lo.ri^{c3}	multu.lo.ri^{c3}
27	11011	mult.hi.rr^{c3}	multu.hi.rr^{c3}	mult.hi.ri^{c3}	multu.hi.ri^{c3}
28	11100	div.q.rr^{c3}	divu.q.rr^{c3}	div.q.ri^{c3}	divu.q.ri^{c3}
29	11101	div.r.rr^{c3}	divu.r.rr^{c3}	div.r.ri^{c3}	divu.r.ri^{c3}
30	11110			lli.i5	lli.i16
31	11111				aui.cr0i16

Table 5.1: Scale compute-op opcodes. A `.rr` extension indicates two register operands, while a `.ri` extension indicates that the second operand is a 5-bit immediate. A `.p` extension indicates that the compute-op targets the predicate register as a destination, and a `.f` extension indicates that the compute-op does a fetch. The superscript cluster identifiers indicate that the opcode is only valid on the given cluster.

ops. The VP instructions in an AIB are processed in order. For each VP instruction, the main RISC operation becomes a compute-op for the associated cluster. This typically includes two local source registers, a local destination register, an opcode, and a predication field. A special destination register index signifies that a compute-op has no local destination. A transport-op attached to the compute-op contains a bit-vector indicating which external clusters are targeted, if any. For each remote write, a writeback-op is generated for the destination cluster. The writeback-op contains a source cluster specifier and a local destination register index. Table 5.2 shows how the Scale software register names are encoded in Scale hardware micro-ops. Figure 5-5 shows a few example AIBs translated into micro-ops.

All inter-cluster data dependencies are encoded by the transport-ops and writeback-ops in the sending and receiving clusters respectively. This allows the micro-op bundles for each cluster to

		compute-op			writeback-op	SD compute-op	SD writeback-op
Scale Reg.	binary encoding	src0	src1	dst	dst	src	dst
`pr0–pr14`	`0–14`	√	√	√	√		
`sr0–sr13`	`31–18`	√	√	√	√		
`prevVP`	`15`	√	√				
null	`15`			√			
`p`	`15`				√		
`cr0`	`16`	√		√	√		
`cr1`	`17`		√	√	√		
`sd0–sd14`	`0–14`					√	√
`sdp`	`15`						√
`sdc`	`16`					√	√

Table 5.2: Binary encoding of Scale register names. The check marks show which micro-op fields each register can appear in.

rgbycc_aib (Scale assembly code shown in Figure 4-2):

SD		C0			C1			C2			C3		
wb-op	compute-op	wb-op	compute-op	tp-op	wb-op	compute-op	tp-op	wb-op	compute-op	tp-op	wb-op	compute-op	tp-op
c1→sd0					c2→cr0	addu cr0,sr0→cr0		c3→cr0				mulh pr0,sr0	→c2
c1→sd1						srl cr0, 16	→sd	c3→cr1	addu cr0,cr1→cr0			mulh pr1,sr1	→c2
c1→sd2					c2→cr0	addu cr0,sr1→cr0		c3→cr1	addu cr0,cr1	→c1		mulh pr2,sr2	→c2
						srl cr0, 16	→sd	c3→cr0				mulh pr0,sr3	→c2
					c2→cr0	addu cr0,sr1→cr0		c3→cr1	addu cr0,cr1→cr0			mulh pr1,sr4	→c2
						srl cr0, 16	→sd	c3→cr1	addu cr0,cr1	→c1		mulh pr2,sr5	→c2
								c3→cr0				mulh pr0,sr6	→c2
								c3→cr1	addu cr0,cr1→cr0			mulh pr1,sr7	→c2
								c3→cr1	addu cr0,cr1	→c1		mulh pr2,sr8	→c2

vtu_decode_ex (Scale assembly code shown in Figure 4-6):

SD		C0			C1			C2			C3		
wb-op	compute-op	wb-op	compute-op	tp-op	wb-op	compute-op	tp-op	wb-op	compute-op	tp-op	wb-op	compute-op	tp-op
c2→sd0			sll pr0,2→cr1		c0→cr0	addu cr0,pVP→cr0		c3→cr0	addu cr0,pVP→cr0		c0→cr0		
			lw cr1(sr0)	→c1		slt cr0,sr0→p			slt cr0,sr0→p		c0→cr1	mulh cr0,cr1	→c2
			copy pr0	→c3		psel cr0,sr0→sr2			psel cr0,sr0→sr2				
		c1→cr0	sll cr0,2→cr1			slt sr1,sr2→p			slt sr1,sr2→p				
			lw cr1(sr1)	→c3		psel sr2,sr1	→nVP,c0		psel sr2,sr1	→sd,nVP			

vpadd_ex:

```
  .aib begin
  c0 lw   sr0(pr0)  -> c1/cr0
  c0 lw   sr1(pr0)  -> c1/cr1
  c1 addu cr0, cr1  -> cr0
  c1 srl  cr0, 2    -> c0/sdc
  c0 sw   sdc, sr2(pr0)
  .aib end
```

SD		C0			C1			C2			C3		
wb-op	compute-op	wb-op	compute-op	tp-op	wb-op	compute-op	tp-op	wb-op	compute-op	tp-op	wb-op	compute-op	tp-op
c1→sdc	sw sdc		lw sr0(pr0)	→c1	c0→cr0								
			lw sr1(pr0)	→c1	c0→cr1	addu cr0,cr1→cr0							
			sw sr2(pr0)			srl cr0, 2	→sd						

Figure 5-5: Cluster micro-op bundles for example AIBs. The writeback-op field is labeled as 'wb-op' and the transport-op field is labeled as 'tp-op'. The prevVP and nextVP identifiers are abbreviated as 'pVP' and 'nVP'.

be packed together independently from the micro-op bundles for other clusters. Therefore, as the Scale loader translates the VP instructions in an AIB into micro-ops, it maintains separate ordered sequences of micro-op bundles for each cluster. As it processes each VP instruction in order, the loader appends the compute-op/transport-op pair and the writeback-ops to the micro-op sequences for the appropriate clusters. The effect of Scale's micro-op bundle packing can be seen by comparing the RGB-to-YCC AIB in Figure 5-5 to the VLIW encoding in Figure 4-3.

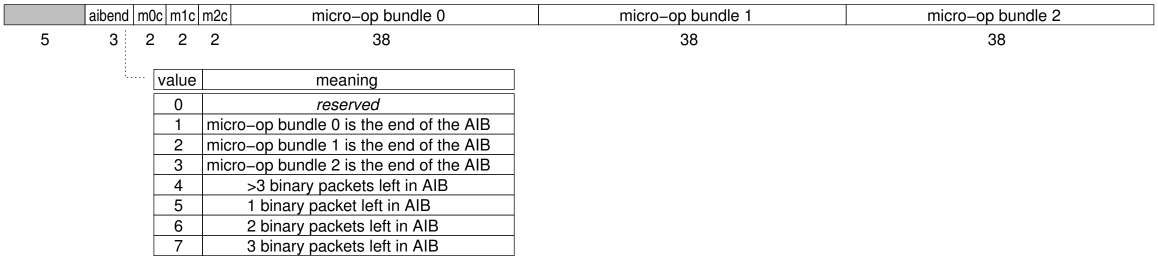
Some Scale VP instructions are handled specially in the conversion to micro-ops. Load-immediate instructions with immediates larger than 5 bits expand into multiple compute-ops and make use of a special micro-op bundle format with 16-bit immediates. Atomic memory instructions expand into a load-lock/op/store-unlock sequence of compute-ops. Long-latency instructions like loads, multiplies, and divides use writeback-ops even for local register destinations. This reduces the bypass and interlock logic complexity in the implementation. Store instructions are split into a store-address compute-op on cluster 0 and a store-data compute-op on the store-data cluster. Regular instructions which target store-data registers generate writeback-ops for the store-data cluster. This c0/sd partitioning enables store decoupling in the implementation.

The semantics of Scale's micro-op bundles dictate that the writeback-op occurs "before" the compute-op when there are read-after-write or write-after-write data dependencies. As the loader builds micro-op bundle sequences for each cluster, whenever possible it merges a compute-op/transport-op pair with a writeback-op from a previous VP instruction. We considered adding a "before/after bit" to the writeback-ops to make the micro-op bundle merging more flexible. However, it is typical for a writeback-op to be followed by a compute-op which uses the data. Optimizing the encoding for this common case saves a critical micro-op encoding bit and simplifies the cluster bypassing and interlock logic.

When VP instructions are decomposed and rearranged into micro-op bundles, an important requirement is that the new encoding does not introduce any possibility of deadlock. The execution semantics of the Scale micro-op encoding requires that the clusters execute their micro-op bundles in parallel, but it does not require any buffering of transport data or writeback-ops. A deadlock-free execution is guaranteed based on the observation that all of the micro-ops for each original VP instruction can execute together to reconstruct the original VP instruction sequence. An implementation is free to pipeline the cluster execution and decouple clusters with queues, as long as it does not introduce any circular dependencies. The micro-ops from the oldest VP instruction must always be able to issue.

Scale's micro-op bundles pair one writeback-op with one compute-op. We considered providing two writeback-ops per compute-op as an alternative arrangement, however we found that one external operand per compute operation is usually sufficient. Many operands are scalar constants or loop invariants, many operands are re-used multiple times, and many operands are produced by local computations. Limiting the number of writeback-ops in a micro-op bundle to one increases encoding density and significantly reduces complexity in the cluster hardware. We also observe that, including the local compute-op destinations, the encoding provides an average of two register writebacks per compute operation. This is more than machines typically provide, but Scale uses clusters to provide this bandwidth without undue complexity – each cluster register file has only two write ports.

An advantage of the Scale micro-op bundle encoding is that a single compute-op can target multiple external destinations. The 6-bit transport-op indicates which external clusters are targeted (clusters-0–3, the store-data cluster, and nextVP). Then, multiple writeback-ops are used to receive the data. The key feature which makes the encoding efficient is that the allocation of writeback-ops is flexible and not tied to compute-ops. On average, there is usually less than one external destination per compute-op, but allowing compute-ops to occasionally target multiple destinations



value	meaning
0	*reserved*
1	micro−op bundle 0 is the end of the AIB
2	micro−op bundle 1 is the end of the AIB
3	micro−op bundle 2 is the end of the AIB
4	>3 binary packets left in AIB
5	1 binary packet left in AIB
6	2 binary packets left in AIB
7	3 binary packets left in AIB

Figure 5-6: Binary instruction packet encoding. Three micro-op bundles are packed together in a 128-bit word. The `m0c`, `m1c`, and `m2c` fields indicate which cluster each micro-op bundle is for. The `aibend` field indicates how many packets remain in the AIB, or which micro-bundle slot in this packet is the end of the AIB.

can avoid copy instructions and significantly boost performance.

Scale's transport/writeback micro-op partitioning is in contrast to clustered VLIW encodings which allow a cluster operation to directly read or write a register in another cluster [GBK07, TTG$^+$03]. Although it may seem like a subtle difference, the transport/writeback partitioning is the key instruction set feature which allows a Scale lane to break the rigid lock-step execution of a VLIW machine. A cluster can run ahead and execute its micro-ops while queuing transport data destined for other clusters. A receiving cluster writes back the data into its register file when it is ready. The important point is that the writeback-op *orders* the register write with respect to the compute-ops on the receiving cluster. Scale's decoupled cluster execution is also made possible by the grouping of instructions into explicitly fetched AIBs. In comparison, a decentralized VLIW architecture with regular program counters must synchronize across clusters to execute branches [ZFMS05].

In effect, decoupled cluster execution enables a lane to execute compute micro-ops out of program order with very low overhead. Compared to register renaming and dynamic dependency tracking in an out-of-order superscalar processor, Scale's clusters use decentralized in-order control structures. Furthermore, the transport data queues and cluster register files are simpler and more efficient than the large monolithic register files which support out-of-order execution in a superscalar machine.

5.2.2 Binary Instruction Packets

During execution, the Scale cluster AIB caches contain micro-op bundle sequences in essentially the same format as those constructed by the loader (Figure 5-5). However, in memory the micro-ops are stored in a more compact form. Usually the number of micro-op bundles is not equal across the clusters, in fact it is not uncommon for only one or two clusters to be used in an AIB. In a typical VLIW binary format, nops are used to fill unused instruction slots. The Scale binary encoding reduces the number of memory bits wasted on nops by encoding AIBs as a unit, rather than encoding each horizontal instruction word individually.

The micro-op bundles in an AIB are decomposed into 128-bit binary instruction packets, Figure 5-6 shows the bit-level encoding. An instruction packet has three micro-op bundle slots. Each slot has an associated field to indicate which cluster it is for. This makes the packing flexible, and in some cases all of the micro-op bundles in an instruction packet may even be for the same cluster. The instruction packet also has a field to indicate which micro-op bundle, if any, is the end of the AIB. To aid the AIB fill unit, this field is also used to indicate how many more instruction packets remain in the AIB.

rgbycc_aib:

aibend	m0c	wb-op	compute-op	tp-op	m1c	wb-op	compute-op	tp-op	m2c	wb-op	compute-op	tp-op
> 3	c1	c2→cr0	addu cr0,sr0→cr0		c2	c3→cr0			c3		mulh pr0,sr0	→c2
> 3	sd	c1→sd0			c1		srl cr0, 16	→sd	c2	c3→cr1	addu cr0,cr1→cr0	
> 3	c3		mulh pr1,sr1	→c2	sd	c1→sd1			c1	c2→cr0	addu cr0,sr1→cr0	
> 3	c2	c3→cr1	addu cr0,cr1	→c1	c3		mulh pr2,sr2	→c2	sd	c1→sd2		
> 3	c1		srl cr0, 16	→sd	c2	c3→cr0			c3		mulh pr0,sr3	→c2
3	c1	c2→cr0	addu cr0,sr1→cr0		c2	c3→cr1	addu cr0,cr1→cr0		c3		mulh pr1,sr4	→c2
2	c1		srl cr0, 16	→sd	c2	c3→cr1	addu cr0,cr1	→c1	c3		mulh pr2,sr5	→c2
1	c2	c3→cr0			c3		mulh pr0,sr6	→c2	c2	c3→cr1	addu cr0,cr1→cr0	
end:m2	c3		mulh pr1,sr7	→c2	c2	c3→cr1	addu cr0,cr1	→c1	c3		mulh pr2,sr8	→c2

vtu_decode_ex:

aibend	m0c	wb-op	compute-op	tp-op	m1c	wb-op	compute-op	tp-op	m2c	wb-op	compute-op	tp-op
> 3	c0		sll pr0,2→cr1		c1	c0→cr0	addu cr0,pVP→cr0		c2	c3→cr0	addu cr0,pVP→cr0	
> 3	c3	c0→cr0			sd	c2→sd0			c0		lw cr1(sr0)	→c1
3	c1		slt cr0,sr0→p		c2		slt cr0,sr0→p		c3	c0→cr1	mulh cr0,cr1	→c2
2	c0		copy pr0	→c3	c1		psel cr0,sr0→sr2		c2		psel cr0,sr0→sr2	
1	c0	c1→cr0	sll cr0,2→cr1		c1		slt sr1,sr2→p		c2		slt sr1,sr2→p	
end:m2	c0		lw cr1(sr1)	→c3	c1		psel sr2,sr1	→nVP,c0	c2		psel sr2,sr1→pr0	→nVP

vpadd_ex:

aibend	m0c	wb-op	compute-op	tp-op	m1c	wb-op	compute-op	tp-op	m2c	wb-op	compute-op	tp-op
2	c0		lw sr0(pr0)	→c1	c1	c0→cr0			sd	c1→sdc	sw sdc	
1	c0		lw sr1(pr0)	→c1	c1	c0→cr1	addu cr0,cr1→cr0		c0		sw sr2(pr0)	
end:m0	c1		srl cr0, 2	→sd								

Figure 5-7: Binary instruction packets for example AIBs (from Figure 5-5).

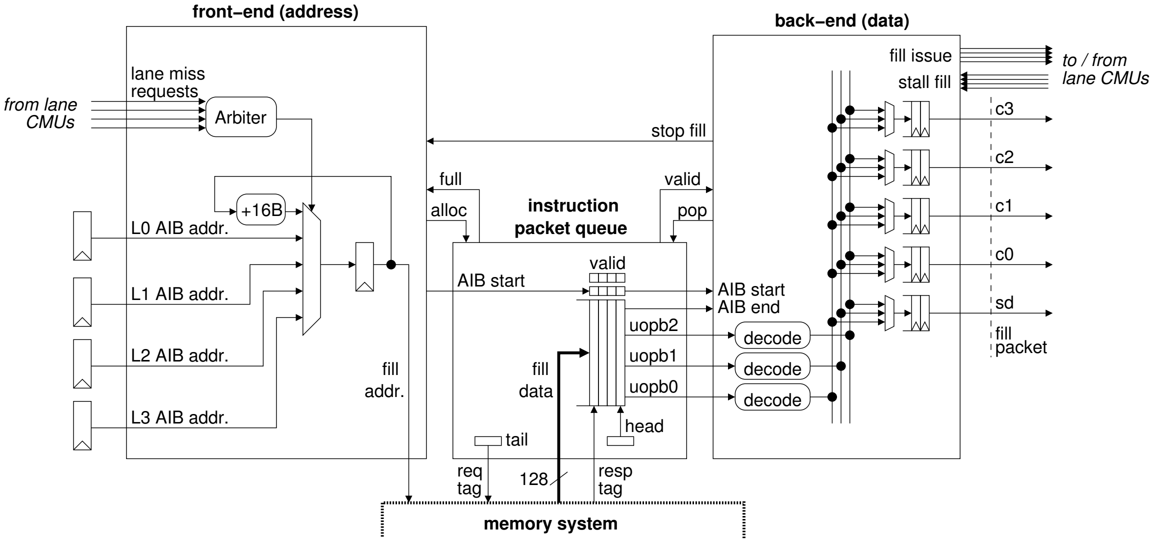


Figure 5-8: AIB fill unit.

The cluster micro-op bundles are organized into instruction packets by striping across the cluster sequences. One micro-op bundle is taken from each cluster before wrapping around, and any clusters with no more micro-op bundles are skipped. For example, if an AIB uses all the clusters, the first instruction packet will have micro-op bundles for c0/c1/c2, the second for c3/sd/c0, the third for c1/c2/c3, and so on. Figure 5-7 shows the binary instruction packet encoding for the example AIBs.

The binary instruction packet encoding does not preclude an AIB from starting in the same packet in which the previous one ends, as long as at most one AIB ends in each instruction packet. However, the current Scale linker uses AIB addresses which are based on the software VP instruction binary format, making it impossible to pack the translated micro-op AIBs next to each other in memory. Therefore, Scale requires all AIBs to begin on naturally aligned 128-bit boundaries. This restriction simplifies the AIB fill unit since AIBs never begin in the middle of an instruction packet.

When AIBs are decomposed into instruction packets, either zero, one, or two micro-op bundle slots are unused at the end of each AIB. The average is one wasted slot per AIB. However, in the current Scale compiler tool chain the software VP instruction encoding is less dense than the instruction packet encoding, so AIBs are spaced apart in memory (Figure 5-2). Thus, out of the six micro-op bundle slots in a 256-bit Scale cache line, between zero and five will be unused at the end of an AIB. This increases the wasted space in the cache to an average of 2.5 micro-op bundle slots per AIB.

We note that the encoding for VP stores is not particularly efficient since each one uses both a store-address micro-op bundle in c0 and a store-data micro-op bundle in sd. We considered encoding these together, but this would make it more difficult for the AIB fill unit to reconstruct the original micro-op bundle sequences in c0 and sd. We also considered packing two store-data micro-op bundles together in one slot. However, given the relatively low frequency of VP stores, we decided to simplify the hardware by using just the simple encoding.

5.2.3 AIB Cache Fill

To service a vector-fetch or a thread-fetch for an AIB that is not cached, the lane command management units send requests to the AIB fill unit. The fill unit fetches AIB instruction packets from memory and reconstructs the per-cluster micro-op bundle sequences. It constructs fill packets with one micro-op bundle for each cluster and broadcasts these to the lane AIB caches across a wide bus. Refilling all of the cluster AIB caches in parallel with wide fill packets simplifies the AIB cache management, allowing the CMU to manage a single AIB cache refill index for all of the clusters in the lane.

An overview of the AIB fill unit is shown in Figure 5-8. The front-end of the fill unit chooses between lane thread-fetch requests using a round-robin arbiter. When all the lanes are fetching the same AIB for a vector-fetch, the fill unit waits until they are all ready to handle the refill together. To accomplish this, priority is given to thread-fetch requests since these come from earlier VTU commands. The lane CMUs will eventually synchronize on each vector-fetch.

After it selects an AIB address, the fill unit begins issuing cache requests for sequential 128-bit instruction packets. It uses a queue to reserve storage for its outstanding requests and to buffer responses which may return out of order. The instruction packet queue slots are allocated in order, and a request is tagged with its chosen slot. When the data eventually returns from the cache, after either a hit or a miss, it is written into its preassigned slot. The fill unit processes the instruction packets in order from the head of the queue. The instruction packet queue needs at least a few entries to cover the cache hit latency, and increasing the capacity allows the fill unit to generate more cache misses in parallel. Scale's queue has 4 slots.

The back-end of the fill unit contains a two-entry queue of micro-op bundles for each cluster. Each cycle it examines the three micro-op bundles at the head of the instruction packet queue, and it issues these into the per-cluster queues. If more than one micro-op bundle in the packet targets a single cluster, the packet will take several cycles to process. To reconstruct wide fill packets with one micro-op bundle per cluster, the fill unit relies on the micro-op bundles being striped across the clusters in the instruction packets. Whenever any cluster queue contains two micro-op bundles, the fill unit can issue all of the head entries in parallel to the destination lane(s). Additionally, when the end of the AIB is reached, the fill unit issues the last group of micro-ops. As it reconstructs and issues fill packets, the fill unit must stall whenever the AIB caches in the destination lane or lanes do not have a free refill slot available.

Although it is possible to construct a fill unit with only one micro-op bundle buffer slot per cluster instead of two, the two-slot design avoids stall cycles when the micro-ops in a single instruction packet are split across two issue groups. We also note that the head micro-op bundle buffer slot serves as a pipeline register, and it could be eliminated at the cost of increased delay within the clock cycle.

An important concern in the design of the fill unit is how to handle the end of an AIB. The front-end runs ahead and fills the instruction packet queue, but since it does not know the size of the AIB it will continue to fetch past the end. As a baseline mechanism, the back-end eventually detects the end of the AIB and it signals the front-end to stop fetching instruction packets. Additionally, all of the extra instruction packets which have already been fetched must be discarded (and some of these may still be in flight in the memory system). To accomplish this, the front-end always tags the beginning of an AIB with a special bit in the instruction packet queue entries. Thus, after reaching the end of an AIB, the back-end discards instruction packets from the queue until it encounters the beginning of a new AIB.

With the baseline AIB end detection mechanism, the front-end will fetch several unused instruction packets. To improve this, the Scale binary instruction packet format uses the AIB-end

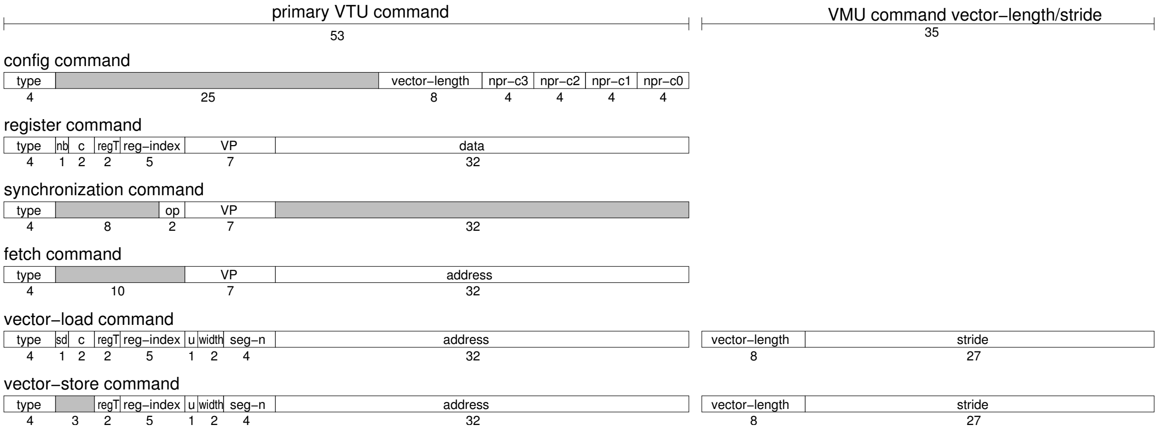


Figure 5-9: Vector-thread unit commands issued by the control processor. Vector memory commands include vector length and stride fields in addition to the primary command fields.

field (which indicates which micro-op in the packet, if any, is the end of the AIB) to also encode how many instruction packets remain in the AIB: 1, 2, 3, or more than 3 (Figure 5-6). The back-end examines this field in the head instruction packet and also considers the number of already allocated instruction packet queue slots. It signals the front-end to stop fetching packets as soon as it determines that the last packet has been fetched. We note that this early stop is precise for large AIBs, but for small AIBs the front-end may still fetch one or two unused instruction packets. In this case correct operation relies on the baseline mechanism.

With the early stop mechanism in place, the front-end of the fill unit is able to begin fetching the next AIB while the back-end continues to process the active one. This can improve performance for code with independent VP threads that require relatively high AIB fetch bandwidth. Ideally, the fill unit could fetch one 128-bit instruction packet (three micro-op bundles) per cycle. The Scale implementation actually has one unused fetch cycle between AIBs due to pipelining. However, this stall cycle in the front-end overlaps with stall cycles in the back-end when the last instruction packet in the AIB has more than one micro-op bundle destined for a single cluster.

A final comment on the AIB fill unit is that it partially decodes the cluster micro-op bundles before inserting them into the AIB caches. This is done to simplify and optimize the cluster control logic. For example, the 5-bit register indices are expanded into a 7-bit format. The top two bits are used to indicate whether the operand is in the register file, a chain register, a predicate register (for destinations only), from the previous-VP (for sources only), or null.

5.3 Command Flow

Commands direct the execution of the lanes and the vector memory units in the VTU. These include VTU commands which originate as control processor instructions, and thread-fetch commands from the lanes themselves. The lane command management units handle AIB caching and propagate commands to the clusters as compact execute directives. The CMUs are also responsible for the tricky task of maintaining correct execution ordering between the VP threads and commands from the control processor.

5.3.1 VTU Commands

A summary of Scale's VTU commands is shown in Tables 4.1 and 4.2. The control processor processes commands and formats them for the VTU as shown in Figure 5-9. The commands are then buffered in a queue, allowing the control processor to run ahead in the program execution. VTU commands can contain many bits since they contain data fields which are read from control processor registers. To improve efficiency, the stride and vector length portions of vector memory commands (35-bits) are held in a separate structure with fewer entries than the main VTU command queue (53-bits). In Scale, the VTU command queue can hold 8 commands of which 4 can be vector memory commands.

At the head of the unified VTU command queue, commands are processed and distributed to further queues in the vector load unit (VLU), the vector store unit (VSU), and the lanes. Vector load and store commands are split into two pieces: an address command is sent to the VLU or VSU and a data command is sent to the lanes.

The VP-configuration and vector length commands are tightly integrated with scalar processing, and they even write the resulting vector length to a CP destination register. Therefore these commands are executed by the control processor, and the CP holds the VTU control registers. The CP contains hardware to perform the maximum vector length calculation – given private and shared register counts it computes the maximum vector length for each cluster, then it calculates the minimum across the clusters. It also sets the new vector length based on the requested length and vlmax. The CP attaches the current vector length to all vector-load and vector-store commands, and it issues an update command to the lanes whenever the VP-configuration or the vector length change.

VTU synchronization commands (`vsync` and `vfence`) stall at the head of the VTU command queue, preventing subsequent commands from issuing until the lanes and the vector memory units are idle. The only difference between a vsync and a vfence is whether or not the control processor also stalls. A VP synchronization (`vpsync`) works like a `vsync` except it only stalls the control processor until the lane that the specified VP maps to is idle.

System-level software on the control processor can issue a `vkill` command to handle exceptions, support time-sharing, or recover from VTU deadlock caused by illegal code. The CP first asserts a signal which stops the lanes and the vector memory units from issuing memory requests, and then it waits for any loads to return. Once the outstanding memory requests have resolved, the control processor resets the entire VTU to complete the `vkill`.

5.3.2 Lane Command Management

Each lane's command management unit contains a single-entry queue to buffer one command at a time. The CMU converts commands into execute directives that it issues to the clusters. Execute directives are queued in the clusters, allowing the CMU to continue processing subsequent commands. Aside from serving as a way point between the control processor and the clusters, the CMU has two main roles: (1) it manages the AIB caches within the lane, and (2) it buffers pending VP thread-fetches and maintains correct ordering between these and the control processor commands.

The structure of an AIB cache is different than a regular instruction cache because AIBs are composed of a variable number of micro-op bundles. Also, the micro-ops in an AIB are packed into instruction packets in the program binary, but in the VTU they are unpacked into a per-cluster organization. The cache management is based on the wide fill packets constructed by the AIB fill unit. These have one micro-op bundle per cluster, allowing the CMU to maintain a single set of AIB cache tags which correspond to these cross-cluster slices. With 32 micro-op bundle slots in each cluster's AIB cache, the CMU contains 32 AIB cache tags in a fully-associative CAM structure.

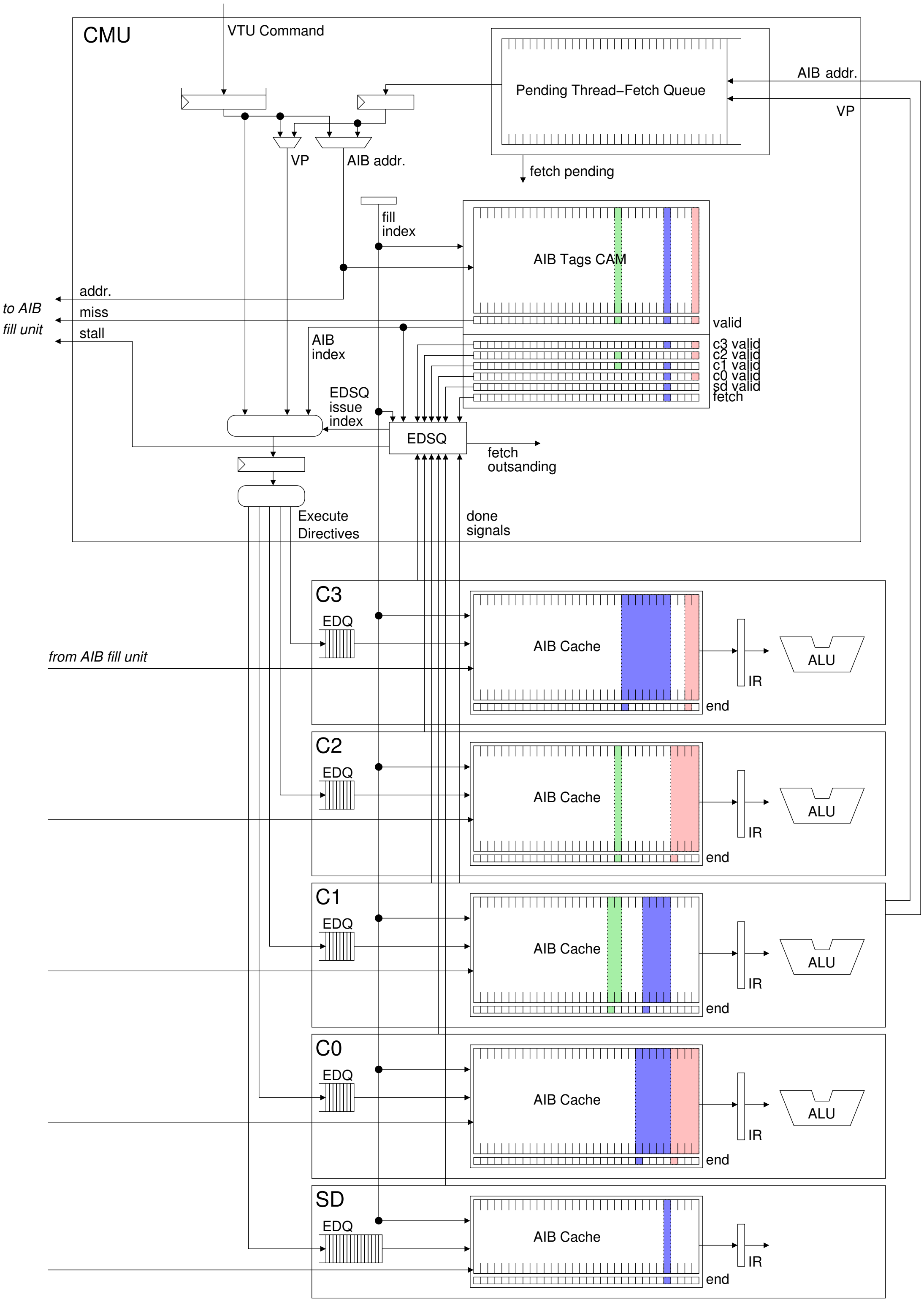


Figure 5-10: Lane command management unit and cluster AIB caches. An example of three cached AIBs is shown.

Each tag includes the AIB address and a valid bit, and extra information bits indicating whether each cluster contains any micro-op bundles in the AIB and whether the AIB contains any thread-fetch instructions. An AIB occupies sequential micro-op bundle slots in the cluster AIB caches and AIBs always start in the same slot across the clusters. The first micro-op slot for an AIB will have a valid tag, and the other tag slots will be unused. The 32 tags allow an AIB to begin in any micro-op bundle slot, and they allow up to 32 (small) AIBs to be cached at the same time. An alternative design could restrict the starting points, e.g. to every other micro-op bundle slot, and thereby require fewer tags.

The AIB caches are filled using a FIFO replacement policy with a refill index managed by the CMU. During each fill cycle, the CMU broadcasts the refill index to the clusters and the AIB fill unit sends a bit to each cluster indicating whether or not it has a valid micro-op bundle. The CMU writes the address of the AIB into the tag corresponding to its first micro-op bundle slot. It sets the valid bit for this tag during the first fill cycle, and it clears the valid bits for the tags corresponding to subsequent fill cycles. The AIB fill unit scans the micro-op bundles during the refill, and on the last fill cycle the CMU updates the additional information bits that are associated with each AIB tag.

Although they have the same starting slot across the clusters, AIBs have a different ending slot in each cluster. To track this, each cluster maintains a set of end bits which correspond to the micro-op bundle slots. During each fill cycle in which a micro-op bundle is written into its AIB cache, the cluster sets the corresponding end bit and clears the previous end bit (except it does not clear an end bit on the first fill cycle). As a result, only the last micro-op bundle in the AIB will have its end bit marked.

When processing a fetch command, the CMU looks up the address in the AIB cache tags. If it is a hit, the match lines for the CAM tags are encoded into a 5-bit index for the starting micro-op bundle slot. This AIB cache index is part of the execute directive that the CMU sends to each cluster which has valid micro-op bundles for that AIB (as indicated by the extra tag bits). When a fetch command misses in the AIB tags, the CMU sends a refill request to the AIB fill unit.

The VT architecture provides efficient performance through decoupling—execute directive queues allow clusters to execute AIBs at different rates and allow the CMU to run ahead in the program stream and refill the AIB caches. However, during a refill the CMU must avoid overwriting the micro-op bundle slots for any AIB which has outstanding execute directives in the clusters. To track which AIB slots are in use, the CMU uses an execute directive status queue (EDSQ) as shown in Figure 5-11. Whenever it issues a fetch ED to the lane, the CMU writes the AIB cache index and the cluster valid bits (as AIB cache active bits) to a new EDSQ entry. The EDSQ entries are allocated in FIFO order, and the chosen EDSQ index is sent to the clusters as part of each fetch ED. When a cluster finishes processing a fetch ED, the CMU uses the EDSQ index to clear the associated AIB cache active bit in that EDSQ entry. An EDSQ entry becomes invalid (and may be reused) when all of its active bits are cleared. Scale has 4 EDSQ entries in each CMU, allowing up to 4 fetch EDs at a time to be executing in the lane. During an AIB cache refill, the CMU broadcasts the fill index to all of the EDSQ entries. If any entry has a matching AIB cache index and any active bits set, the CMU stalls the AIB fill unit. This mechanism will stall the fill unit as soon as it reaches the first slot of any active AIB, and since the AIB caches are filled in FIFO order this is sufficient to avoid overwriting any active AIB slots.

A defining feature of a VT architecture is that a VP executing a vector-fetched AIB can issue a thread-fetch to request its own AIB. These are buffered in the CMU in a pending thread-fetch queue (PTFQ). A PTFQ entry includes an AIB address and the requesting VP index number. Although a VT ISA does not necessarily dictate the order in which thread-fetches for different VPs are processed, Scale uses a simple FIFO structure for the PTFQ to provide fairness and predictability. The PTFQ must be able to buffer one pending fetch for each VP mapped to a lane, and Scale supports up

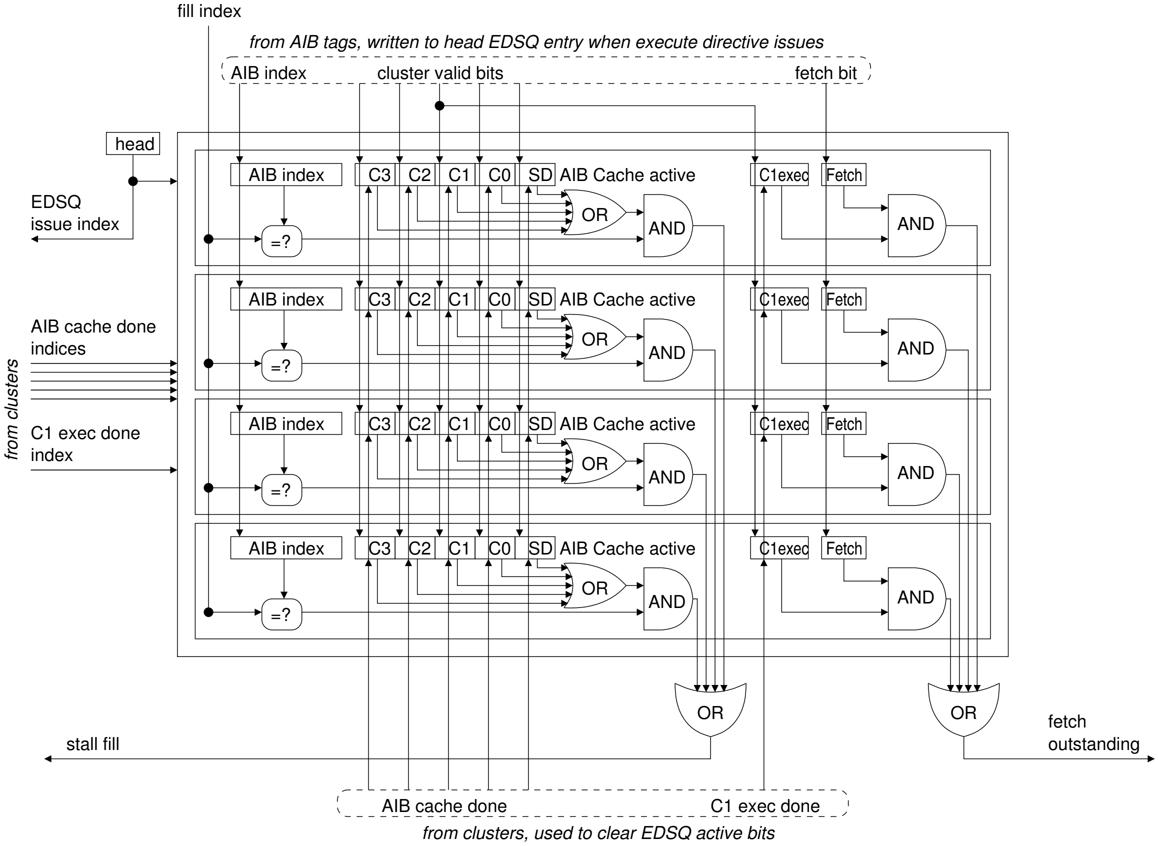
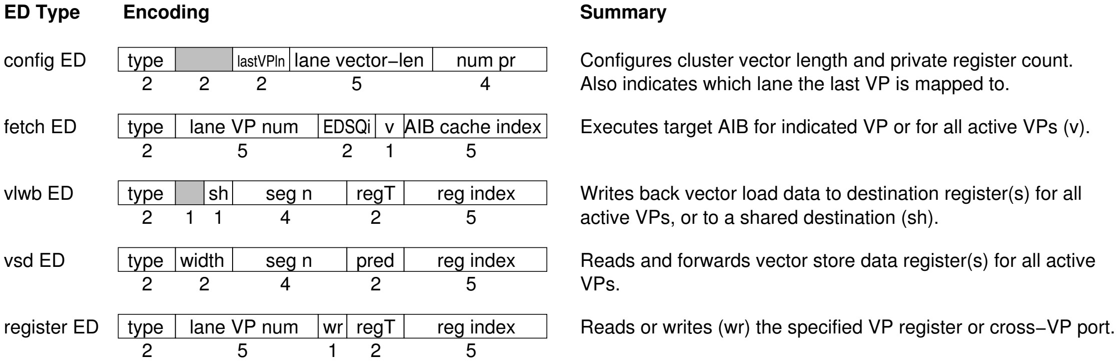


Figure 5-11: Execute Directive Status Queue.

to 32 VPs per lane. Scale VPs are always fetch-capable, but an alternative design could reduce the number of PTFQ entries and limit the vector length when the VPs are configured as fetch-capable.

With vector and thread fetches operating concurrently in the VTU, an important consideration is the interleaving between these VP control mechanisms. The VT execution semantics dictate that a VP thread launched by a vector-fetch should execute to completion before being affected by a subsequent control processor command. Therefore, Scale always gives thread-fetches in the PTFQ priority over VTU commands from the CP. However, before it processes the next VTU command the CMU must also ensure that none of the buffered execute directives will issue a thread-fetch. To accomplish this, additional bits in each EDSQ entry indicate if the AIB contains a fetch and if cluster 1 (the cluster which can execute fetch instructions) has finished executing the AIB. The fetch bit is set based on the information in the associated AIB cache tag, and the c1-exec bit is handled in the same way as the AIB cache active bits. The CMU will not process a VTU command from the CP until (1) there are no pending thread-fetches in the PTFQ, and (2) there are no possible thread-fetches in outstanding execute directives. The one exception to this rule is that Scale provides non-blocking individual VP read, write, and fetch commands which are given priority over thread-fetches on the lane.



ED Type	Encoding	Summary
config ED	type (2), [shaded] (2), lastVPln (2), lane vector−len (5), num pr (4)	Configures cluster vector length and private register count. Also indicates which lane the last VP is mapped to.
fetch ED	type (2), lane VP num (5), EDSQi (2), v (1), AIB cache index (5)	Executes target AIB for indicated VP or for all active VPs (v).
vlwb ED	type (2), [shaded] (1), sh (1), seg n (4), regT (2), reg index (5)	Writes back vector load data to destination register(s) for all active VPs, or to a shared destination (sh).
vsd ED	type (2), width (2), seg n (4), pred (2), reg index (5)	Reads and forwards vector store data register(s) for all active VPs.
register ED	type (2), lane VP num (5), wr (1), regT (2), reg index (5)	Reads or writes (wr) the specified VP register or cross−VP port.

Figure 5-12: Cluster execute directives.

5.3.3 Execute Directives

VTU commands issue from the control processor, propagate through the lane command management units, and the CMUs eventually issue the commands to the individual clusters in the form of execute directives. The format and encoding for cluster execute directives is shown in Figure 5-12. Each cluster contains an execute directive queue (EDQ), and the CMU usually issues an ED to all the clusters in its lane at the same time. In Scale the cluster EDQs have 8 entries, except for the store-data cluster which has 16 entries to improve store decoupling.

A VP-configuration or vector length command turns into a *config ED* for the VTU clusters. A config ED indicates how many active VPs are mapped to the lane and the number of private registers each VP has in that cluster. To aid with cross-VP transfers, each cluster is also informed about which lane the last VP is mapped to. The CMU also tracks the vector length commands, and when there are zero VPs mapped to a lane it will ignore certain VTU commands.

A vector-fetch or VP-fetch from the control processor turns into a *fetch ED*, as does a thread fetch generated by a VP. A fetch ED identifies the target AIB with an index into the AIB cache. It contains a flag to indicate that the AIB should be executed by all active VPs (for a vector-fetch), or otherwise it identifies the particular VP index number which should execute the AIB. Finally, a fetch ED contains an EDSQ index to help the CMU track the status of outstanding fetch EDs.

A vector-load command from the control processor is sent to the vector load unit for processing, and a corresponding a *vector load writeback (vlwb) ED* is sent to the destination cluster. A vlwb ED contains a destination register specifier which includes a register type and an index number. Segment vector loads include a segment size field to indicate the number of elements, this field is set to 1 for regular vector loads. Usually a vlwb ED writes data for every active VP mapped to the lane, but a special flag indicates that it is for a shared vector load and should only execute once.

A vector-store command from the control processor is sent to the vector store unit for processing, and a corresponding *vector store data (vsd) ED* is sent to the store-data cluster. The vsd ED instructs the store-data cluster to forward data to the vector store unit from each active VP mapped to the lane. It includes the source register index, a predication field, the segment size, and the byte-width.

VP register write commands are sent to the command management unit in the destination lane. The CMU forwards the data to a special queue in cluster 0. It also sends cluster 0 a *register ED* which reads the data from this queue and sends it to the destination cluster (if the destination is not c0 itself). A corresponding register ED sent to the destination cluster receives the data from c0 and writes it to the target register. Cross-VP push commands are handled similarly—they are sent to the last VP, and the destination cluster sends the data out on its nextVP port.

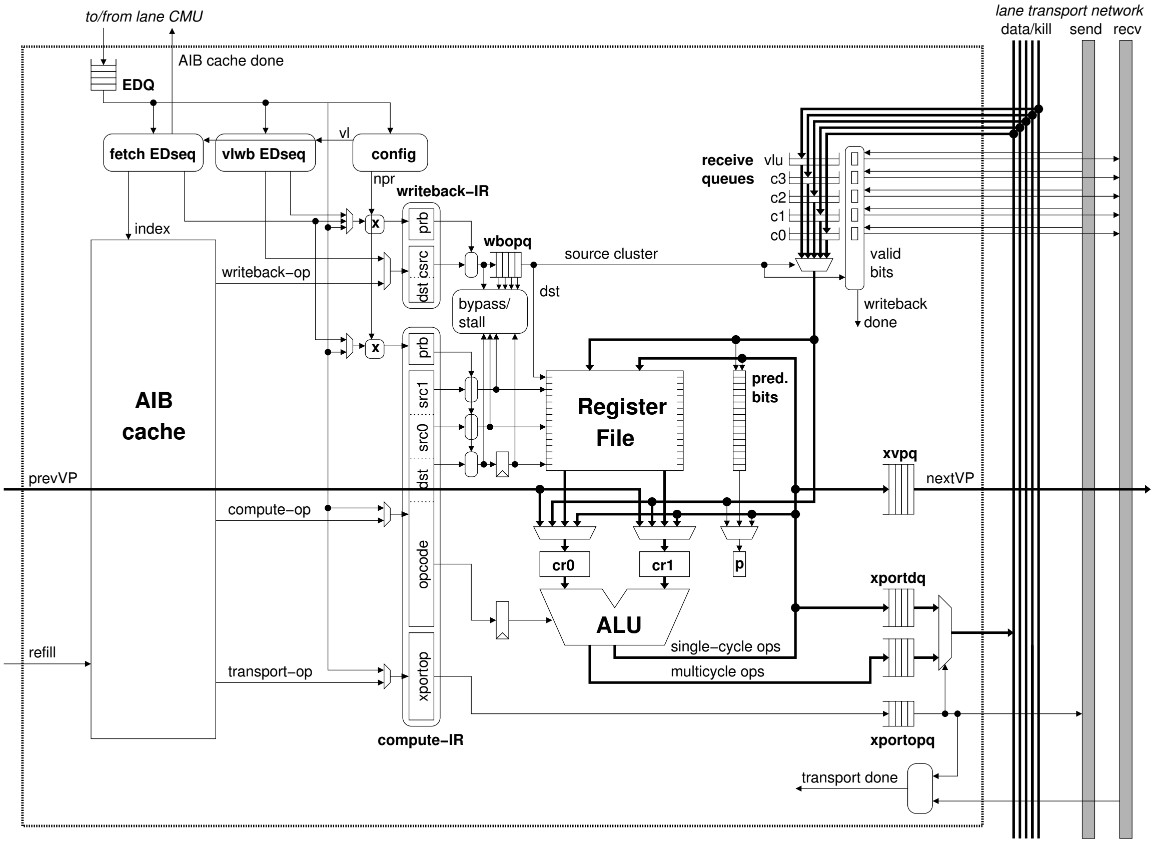


Figure 5-13: Generic cluster microarchitecture.

To handle VP register read commands from the control processor, the CMU sends the source cluster a register ED which forwards the data to cluster 0. The CMU also sends a register ED to cluster 0 which receives the data and forwards it to the control processor. Meanwhile, after it issues a register read command, the control processor stalls until the data is available (only one register read at a time is active in the VTU). Cross-VP pop commands are essentially register reads which execute on VP0 and read data from the source cluster's prevVP port. A cross-VP drop command is similar, except the data is not returned to the control processor.

5.4 Vector-Thread Unit Execution

The execution resources in the vector-thread unit are organized as lanes that are partitioned into independent execution clusters. This section describes how clusters process execute directives from a lane's command management unit, and how data is communicated between decoupled clusters. This section also describes how the memory access cluster and the store-data cluster provide access/execute decoupling, and it discusses the operation of the cross-VP start/stop queue.

5.4.1 Basic Cluster Operation

Each cluster in the VTU operates as an independent execution engine. This provides latency tolerance through decoupling, and it reduces complexity by partitioning control and data paths. Fig-

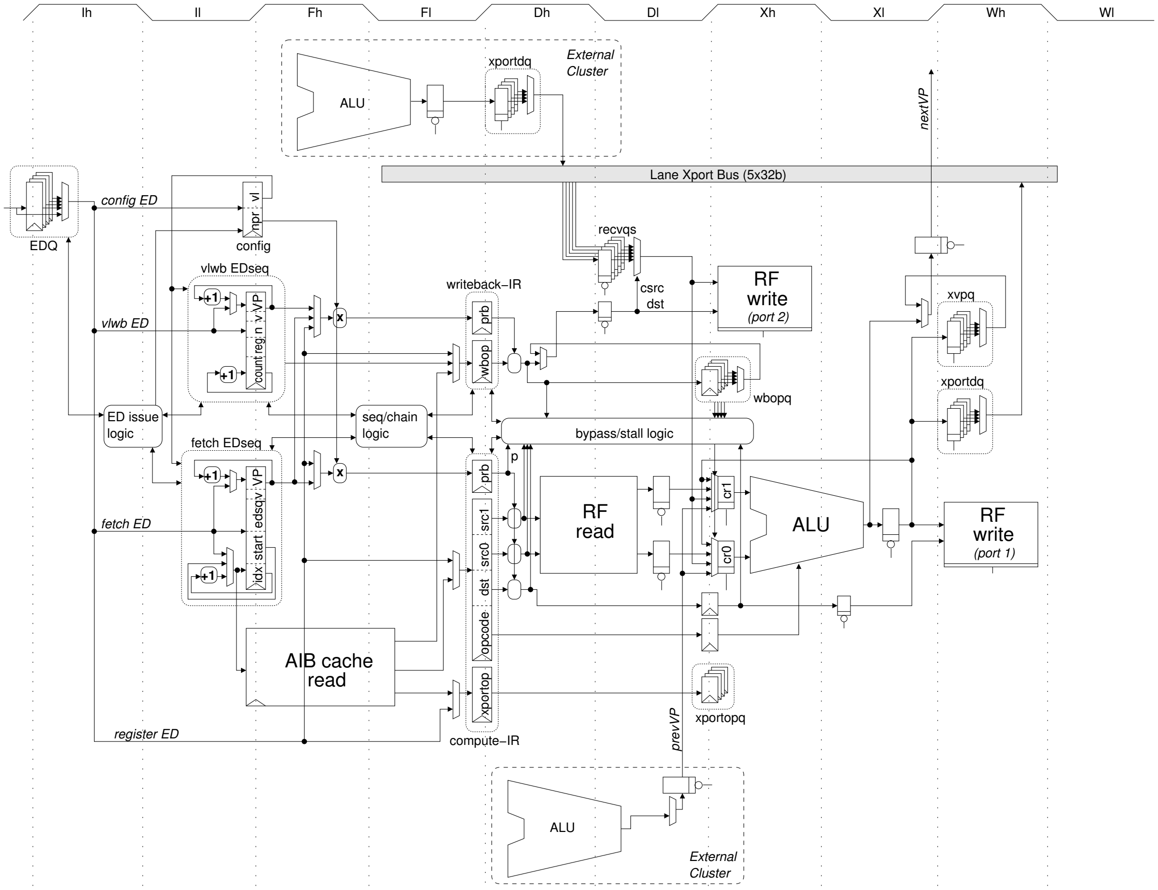


Figure 5-14: Cluster pipeline.

ure 5-13 shows the general structure of a cluster, and Figure 5-14 shows the pipeline diagram. A cluster contains a register file and an arithmetic and logical unit (ALU), and these resources are used to hold VP registers and execute micro-ops. Micro-ops are held in two instruction registers—the writeback-IR holds a writeback-op and the compute-IR holds a compute-op and a transport-op. A cluster also has an execute directive queue and an AIB cache. The clusters within a lane are interconnected with a broadcast data network, and a cluster's decoupled transport and writeback units send and receive data to/from other cluster. Clusters also connect to their siblings in other lanes with nextVP and prevVP data ports.

Clusters are ultimately controlled by execute directives and these are processed by *execute directive sequencers*. The *fetch ED sequencer* handles AIB fetches. Cycle-by-cycle it increments its AIB cache index to step through the target AIB's micro-op bundles and issue them for the specified VP. For a vector-fetch, the fetch ED sequencer repeats this process to execute the AIB for every VP mapped to the lane. It initializes the active VP index to 0, and then increments it after each iteration through the AIB. To save energy, Scale eliminates redundant AIB cache reads when a vector-fetched AIB has only one micro-op bundle. The *vector load writeback ED sequencer* repeatedly issues writeback-ops for every VP mapped to the lane. These writeback-ops receive data from the vector load unit and write it back to the cluster register file. For segment vector loads, the sequencer issues multiple writeback-ops for each VP. The store-data cluster has a *vector store data ED sequencer* which issues compute-ops to send store data to the vector store unit. In the most

	binary	VRI		PRI	
Scale Reg.	encoding	type	index	type	index
pr0–14	0–14	RF	0–14	RF	prb+VRI
sr0–13	31–18	RF	31–18	RF	VRI
prevVP	15	P	0	NULL	0
null	15	NULL	0	NULL	0
p	15	P	0	P	prb
cr0	16	CR	0	CR	0
cr1	17	CR	1	CR	1
sd0–14	0–14	RF	0–14	RF	prb+VRI
sdp	15	P	0	P	prb
sdc	16	CR	0	CR	0

Table 5.3: Mapping of Scale virtual register indices (VRIs) to physical register indices (PRIs). The AIB fill unit recodes the 5-bit binary indices as 7-bit VRIs organized by register type. The VRI to PRI translation is made when a micro-op executes for a particular virtual processor, and it takes into account the VP's private register base index (prb). Note that the P VRI type encodes either prevVP for a source operand or the predicate register for a destination operand.

straightforward implementation, only one ED sequencer is active at a time. However, to improve performance, two ED sequencers can chain together and execute simultaneously. ED chaining is described further in Section 5.4.4. Finally, config EDs and register EDs do not use ED sequencers. A config ED updates the number of private registers and the lane vector length when it issues. A register ED issues straight into the compute-IR and writeback-IR.

Resources in the VTU are virtualized, and micro-ops contain virtual register indices (VRIs). When a micro-op executes for a particular virtual processor, the VP number and the VRIs are combined to form physical register indices (PRIs). Table 5.3 shows the mapping. The VRIs for shared registers (31–19) directly map to PRIs. In this way, the shared registers extend downwards from register 31 in the register file. VRIs for private registers are translated by adding the private register index to the VP's base index in the register file (prb). The base index is computed by multiplying a VP's lane index number by the configured number of private registers per VP.

Micro-op bundles execute in a 5-stage pipeline: index (I), fetch (F), decode (D), execute (X), and writeback (W). During the index stage the fetch ED sequencer calculates the index into the AIB cache, this is analogous to the program counter in a traditional processor. Execute directives also issue from the EDQ to the ED sequencers during I. A micro-op bundle is read from the AIB cache during the fetch stage. Also during F, the active VP's base register file index (prb) is calculated. The compute-IR and writeback-IR are updated together at the end of F. At the beginning of the decode stage, virtual register indices are translated into physical register indices. Bypass and stall logic determine if the compute-IR and writeback-IR can issue. The operands for a compute-op may be read from the register file, bypassed from X, bypassed from a writeback, popped from the prevVP port, taken as an immediate from the compute-IR, or already held in a chain-register. Compute-ops go through the ALU in the execute stage, and they write a destination register during the writeback stage.

The cross-VP network links VPs in a ring, and cross-VP dependencies resolve dynamically. A compute-op which uses prevVP as an operand will stall in the decode stage until data is available on this port. When a transport-op targets nextVP, the data is written into the cross-VP queue (xvpq) during the writeback stage. When data is available in the sender's cross-VP queue, and the receiver is ready to use an operand from prevVP, the cross-VP data transfer completes. If the cross-VP queue is empty, the ALU output from the execute stage can be forwarded directly to this port.

This zero-cycle nextVP/prevVP bypass enables back-to-back execution of compute operations in a cross-VP chain. The size of the cross-VP queue determines the maximum number of cross-VP operations within an AIB. In Scale, each cluster's cross-VP queue holds 4 entries. Interaction with the cross-VP start/stop queue is described in Section 5.4.7.

Decoupled transports allow a cluster to continue executing micro-ops when a destination cluster is not ready to receive data. Transport-ops (other than nextVP transports) are enqueued in the transport-op queue (xportopq) after they issue from the compute-IR. Data is written into the transport data queue (xportdq) at the beginning of the writeback stage. When a transport-op reaches the head of the queue, a transport can take place as soon as the data is available. This can occur as early as the writeback stage in the cluster pipeline. When a transport completes, both the transport-op queue and the transport data queue are popped together. Section 5.4.3 describes the transport unit in more detail.

The organization of compute-ops and writeback-ops as a sequence of micro-op bundles defines the relative ordering of their register reads and writes. Within a bundle, the writeback-op precedes the compute-op. A baseline implementation could stall the writeback-IR whenever transport data from an external cluster is unavailable. To improve performance, writebacks are written into the writeback-op queue (wbopq) at the end of the decode stage. At the beginning of this stage the active writeback-op is selected, either from the head of the queue, or from the writeback-IR if the queue is empty. If transport data is available for the active writeback, the register file or chain register is written during the next cycle (X). With writeback-ops buffered in a queue, the compute-op stream can run ahead of the writeback-op stream as long as register dependencies are preserved. The entries in the writeback-IR and the writeback-op queue always precede the compute-IR. At the beginning of the decode stage, the compute-IR register indices (the PRIs) are compared to all of the writeback-op destination register indices. The compute-IR will stall if any of its sources match a writeback, unless the writeback data is available to be bypassed that cycle. The compute-IR must also stall if its destination matches an outstanding writeback (to preserve the write-after-write ordering). Scale's writeback-op queue has 4 entries. The bypass/stall logic includes 20 7-bit comparators to compare 5 writeback-op indices (4 in the queue plus the writeback-IR) to 4 compute-IR indices (two sources, one destination, and the predicate register).

Each VP has a private predicate register and these are held in a 32×1-bit register file indexed by the VPs base register file index (prb). All compute-ops have a predication field which indicates either positive, negative, or no predication. The predicate register is read during the decode stage of the pipeline, or possibly bypassed from a compute-op in X or a writeback-op. If the predication condition is false, the compute-op is killed during D so that it does not propagate down the pipeline. In particular, a compute-op implicitly over-writes the chain registers with its operands when it executes, but a predicated compute-op must not be allowed to modify these registers. We discovered a critical path in Scale which occurs when the predicate register is bypassed from X to determine whether or not a predicated compute-op will overwrite a chain-register that is simultaneously being targeted by a writeback-op. This is a rare condition since a chain-register written by a writeback-op is not usually overwritten by a compute-op. Therefore, Scale stalls compute-ops to eliminate the critical path from a bypassed predicate register to the chain register mux select signals. Another critical path occurs when a predicated compute-op uses prevVP as a source operand—the cross-VP data transfer should only occur if the compute-op actually executes. Again, Scale stalls compute-ops to avoid the critical path from a bypassed predicate register to the ready-to-receive signal for the prevVP port.

5.4.2 Long-latency Operations

Long-latency compute-ops—including multiplies, divides, loads, and floating-point operations—use a separate transport data queue. This avoids the problem of arbitrating between single-cycle and multi-cycle compute-ops for writing into a unified queue. When transport-ops are queued in stage D of the pipeline, they are updated with a special field to indicate their source transport data queue. They then wait at the head of the transport-op queue for data to be available in their source queue. In this way, a cluster emits its external transport data in the correct order, regardless of the latency of its execution units.

Long-latency compute-ops with local destinations have the potential to complicate the cluster bypassing and interlock logic as well as the destination register writeback. The destination register indices for in-flight multi-cycle operations must be checked against the sources and destinations of subsequent compute-ops. Multi-cycle operations must also contend with single-cycle operations for the register file write port. Instead of adding hardware, Scale's long-latency compute-ops use transport-ops and writeback-ops to target local destinations. Then, all of the necessary interlock and bypassing checks, as well as the bypass itself and the register file write, are handled with the existing writeback-op hardware. The extra writeback-op for local destinations is an acceptable cost, especially since long-latency operations usually target remote clusters anyways to improve cluster decoupling. One minor restriction which Scale must impose on long-latency compute-ops is that they are not allowed to directly target nextVP as a destination (an additional copy instruction is required).

In Scale, cluster 3 provides 16-bit multiplies. The multiplier actually completes in a single cycle, but it acts as a long-latency unit to avoid critical paths. This means that if a multiply on c3 is followed by a compute-op which uses its result, the second compute-op will stall for one cycle (assuming that the transport and writeback units are idle). Cluster 3 also has an iterative 32-bit multiplier and divider which shares the transport data queue with the 16-bit multiplier. To handle the contention, only one of these operation types are allowed to be active at a time. Loads on cluster 0 also act as long-latency operations. They use a special transport data queue that is described in more detail in Section 5.4.5.

5.4.3 Inter-cluster Data Transfers

Recalling the VP instruction to micro-op translation, all inter-cluster data communication is split into transport and writeback operations, and every transport has a corresponding writeback. Each cluster processes its transport-ops in order, broadcasting result values to the other clusters in the lane on its dedicated transport data bus. Each cluster also processes its writeback-ops in order, writing the values from external clusters to its local register set. A clock cycle is allocated for the transport and writeback, meaning that dependent compute operations on different clusters will have a one cycle bubble between them (which may be filled with an independent compute operation). An implementation could allow transports destined for different clusters to send data out of order, but the in-order implementation uses one transport data bus per cluster rather than a bus connecting every pair of clusters.

It is sufficient for a baseline implementation to only perform inter-cluster data transfers when both the sender and the receiver are ready. However, this can sometimes cause inefficiencies when the sender is ready before the receiver. In this case the sender's transport unit will stall, possibly delaying subsequent transports destined for other clusters. We have found that it is helpful to provide a small amount of buffering to allow a cluster to receive data from different senders out-of-order with respect to its writeback-ops. Scale clusters have 5 single-entry receive queues, one for each

send	receive	send	receive	send	receive	send	receive	send	receive	kill
C0toC0	C0fromC0	C1toC0	C0fromC1	C2toC0	C0fromC2	C3toC0	C0fromC3	VLUtoC0	C0fromVLU	C0kill
C0toC1	C1fromC0	C1toC1	C1fromC1	C2toC1	C1fromC2	C3toC1	C1fromC3	VLUtoC1	C1fromVLU	C1kill
C0toC2	C2fromC0	C1toC2	C2fromC1	C2toC2	C2fromC2	C3toC2	C2fromC3	VLUtoC2	C2fromVLU	C2kill
C0toC3	C3fromC0	C1toC3	C3fromC1	C2toC3	C3fromC2	C3toC3	C3fromC3	VLUtoC3	C3fromVLU	C3kill
C0toSD	SDfromC0	C1toSD	SDfromC1	C2toSD	SDfromC2	C3toSD	SDfromC3	VLUtoSD	SDfromVLU	

Table 5.4: Send, receive, and kill control signals for synchronizing inter-cluster data transport operations (within a lane). The source cluster asserts the send and kill signals, and the destination cluster asserts the receive signal.

possible sender (the 4 clusters plus the vector load unit). These are implemented as transparent latches, allowing a writeback-op to issue on the same cycle that the corresponding transport data is written into the receive queue (in stage D of the cluster pipeline).

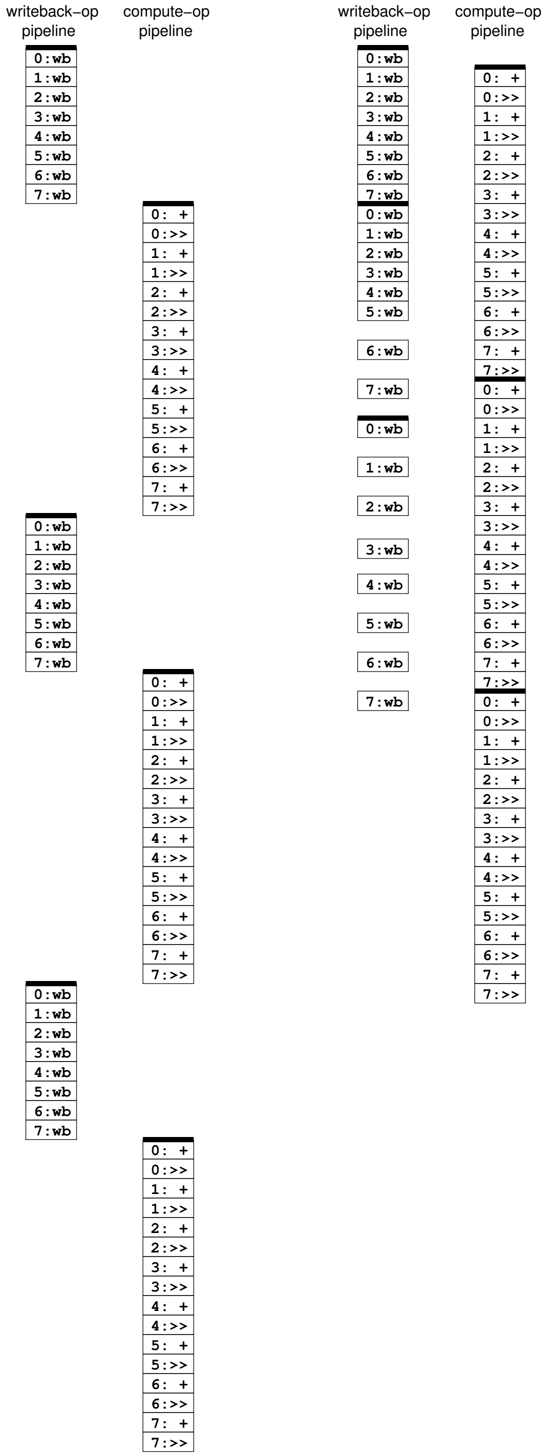
The inter-cluster data transfers are synchronized with *send* and *receive* control signals which extend between each cluster and every other cluster in a lane. A list of the signals is shown in Table 5.4. Each cluster asserts its receive signals based on the state of its receive queues: it is ready to receive data in any queue which is not full. In an implementation with no receive queues, the receive signals would be set based on the source of the active writeback-op (csrc). Each cluster asserts its send signals based on the destination bit-vector in its active transport-op (when the transport data is ready). A data transfer occurs whenever a send signal pairs with a receive signal. The sending and receiving clusters detect this condition independently, avoiding the latency of round-trip communication paths. A multi-destination transport-op completes once all of its sends have resolved.

When a VP instruction executes under a false predicate it produces no result and both its local and remote destination writebacks are nullified. Achieving this behavior is not trivial in an implementation with distributed transports and writebacks. To accomplish it, a kill signal is associated with every transport data bus (one per cluster). When a predicated compute-op is nullified in stage D of the pipeline, a special kill bit is set in the transport-op queue entry. When this transport-op reaches the head of the queue, it does not use the transport data queue. Instead of sending data, it asserts the cluster's kill signal. The send/receive synchronization works as usual, and the kill signal is recorded as a kill bit in the destination cluster's receive queue. When a writeback-op sees that the kill bit is set, it nullifies the writeback.

5.4.4 Execute Directive Chaining

The goal of execute directive chaining is to allow vector-load, vector-fetch, and vector-store commands to overlap. This is similar to chaining in a traditional vector machine, except there is no chaining of regular compute operations—this is unnecessary since clustering already allows multiple compute operations to execute in parallel. The opportunity for chaining arises from the fact that a vector load writeback ED only uses the writeback portion of the cluster pipeline, leaving the compute portion idle. Conversely, many vector-fetch EDs only use the compute portion of the pipeline, leaving the writeback portion idle. Figure 5-15(a) shows the inefficiency when vector-load and vector-fetch commands execute back-to-back in a cluster without chaining.

The foundation for chaining rests on two properties of a vector (or vector-thread) machine: (1) VPs have independent register sets, and (2) vector commands iterate sequentially through the VP vector. Thus, when a vector-fetch follows a vector-load, it does not need to wait for the vector-load to finish before it can start executing. Instead, VP0 can start executing the fetch as soon as the vector load writeback for VP0 is complete, VP1 can execute the fetch as soon as the vector load writeback for VP1 is complete, and so on. The overlapped execution is guaranteed not to violate any



(a) without chaining

(b) with chaining

Figure 5-15: Execute directive chaining. The example scenario includes a vector-load followed by a vector-fetch, repeated three times. The issuing micro-ops in a single cluster are shown, both without (a) and with (b) chaining. The vector-fetched AIB contains an add compute-op (+) followed by a right-shift compute-op (>>). A vector length of 8 is assumed, and lane VP numbers for the issuing writeback-ops and compute-ops are indicated before the colon. Time progresses downwards in the figure.

register dependencies because the registers used by the VP0 fetch can not be written by the VP1–n writebacks, the registers used by the VP1 fetch can not be written by the VP2–n writebacks, etc. The argument is symmetric when a vector-load chains on a previous vector-fetch. Figure 5-15(b) shows how the back-to-back vector-load and vector-fetch example executes with chaining. The first vector-fetch chains on the first vector-load, and then the second vector-load chains on the first vector-fetch. In this example the AIB has two compute-ops, so the second vector-load eventually gets throttled by the vector-fetch progress. Eventually, the steady-state has a vector-load chaining on the previous vector-fetch.

To generalize the concept of execute directive chaining we introduce the notion of a *primary ED sequencer* and a *chaining ED sequencer*. Recall that ED sequencers operate during the fetch stage of the cluster pipeline to issue micro-ops to the instruction registers (the compute-IR and the writeback-IR). A primary ED sequencer is for the older ED and it is always free to issue micro-ops, with the exception that a new primary ED sequencer must wait when there are any micro-ops from a previous ED sequencer stalled in the IRs. A chaining ED sequencer is for the newer ED and it is only allowed to issue micro-ops when (1) its VP number is less than that of the primary ED sequencer and (2) the primary ED sequencer is not stalled. The primary and chaining ED sequencers are free to issue micro-ops together on the same cycle (into different IRs and for different VPs). A chaining ED sequencer becomes primary after the primary ED sequencer finishes.

The remaining question is when a new execute-directive from the EDQ is allowed to issue to a chaining ED sequencer. The foremost restriction is that chaining is only allowed for vector EDs onto other vector EDs. Non-vector EDs such as those from thread-fetch commands are not allowed to chain on other EDs, and other EDs are not allowed to chain on them. Shared vector-loads are also not allowed to chain since they write to shared registers instead of private VP registers. The second restriction on chaining is that the EDs must not issue micro-ops to the same IRs. Vector load writeback (vlwb) EDs only use the writeback-IR and vector store data (vsd) EDs only use the compute-IR, so these EDs are free to chain on each other. Vector-fetch EDs may use both IRs, depending on the AIB. To enable chaining, the fetch ED sequencer records whether or not the AIB has any compute-ops or writeback-ops as it executes for the first time (for VP0 on the lane). Then, a vlwb ED may chain on a vector fetch ED as long as the AIB does not have any writeback-ops, and a vsd ED may chain on a vector fetch ED as long as the AIB does not have any compute-ops. A vector fetch ED may also chain on a vlwb ED or a vsd ED, but in this case the fetch ED sequencer does not yet know what micro-ops are in the AIB. Therefore, when the fetch ED sequencer is chaining on the vlwb ED sequencer it will stall as soon as it encounters a writeback-op in the AIB. Similarly, when the fetch ED sequencer is chaining on the vsd ED sequencer it will stall as soon as it encounters a compute-op in the AIB. In these cases the fetch ED sequencer continues to stall until it becomes the primary ED sequencer.

Since execute directive chaining causes command operations to be processed out-of-order with respect to the original program sequence, deadlock is an important consideration. To avoid deadlock, the oldest command must always be able to make progress—a newer command must not be able to block it. As an example of a potential violation, we were tempted to allow a vector-load to chain on a vector-fetch even if the vector-fetch had some writeback-ops. The idea was to allow the vector-load writebacks to slip into any unused writeback-op slots as the vector-fetch executed. However, this imposes a writeback ordering in which some of the vector-load writebacks must precede some of those from the earlier vector-fetch, and in some situations the vector-load writebacks can block the vector-fetch from making progress. For example, consider a vector-load to c0, followed by a vector-fetched AIB in which c0 sends data to c1, followed by a vector-load to c1. In program order, the vector load unit must send a vector of data to c0, then c0 must send a vector of data to c1, and then the vector load unit must send a vector of data to c1. However, if the vector-load writeback

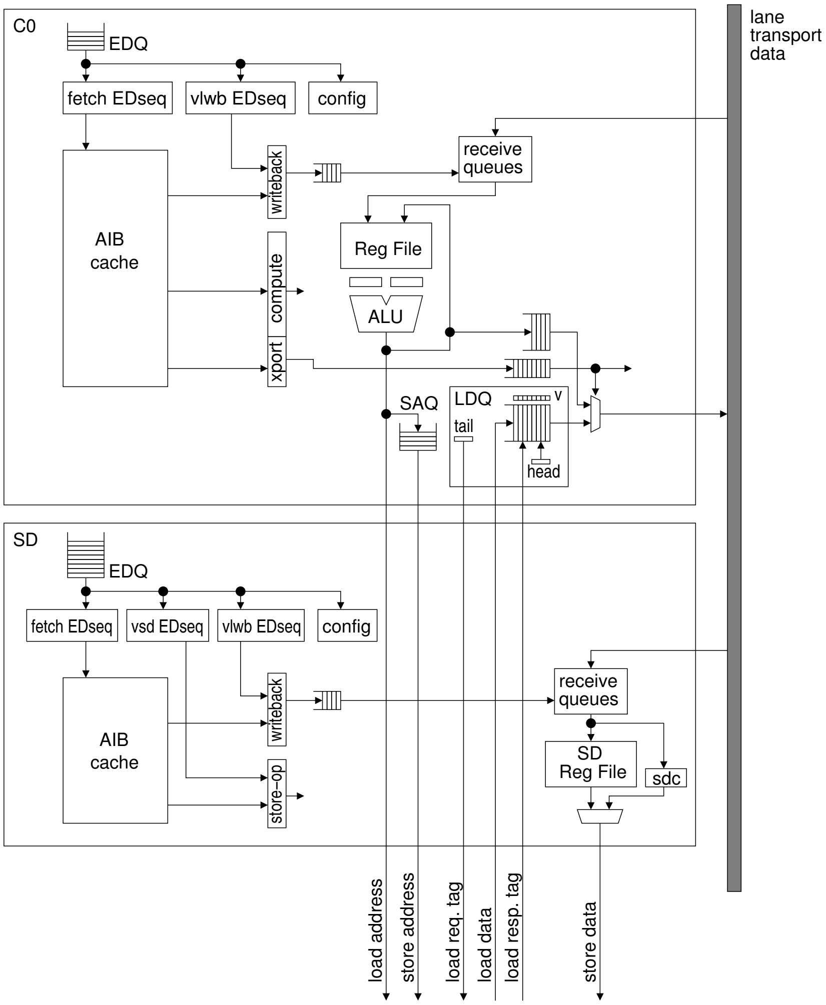


Figure 5-16: Cluster 0 and store-data cluster microarchitecture. The load data queue (LDQ) and store address queue (SAQ) in cluster 0 enable memory access decoupling.

chains on the vector-fetch in c1, it will try to receive data from the vector load unit before receiving *all* of the data from c0. With c1 waiting for data from the vector load unit, c0 will stall trying to send data to c1, and the vector load unit will stall trying to send data to c0, thereby completing the cycle for deadlock. Again, the fundamental problem is that the vector-load writeback-ops on c1 are able to block the progress of the older vector-fetch micro-ops. We avoid this situation by only allowing vector load writebacks to chain on AIBs which do not have any writeback-ops.

5.4.5 Memory Access Decoupling

Indexed accesses are an important category of loads and stores that lack the structure exploited by vector memory commands. Instead, the VPs themselves access data using single-element load and store instructions. Typically, cluster 0 loads data values from memory and sends them to other clusters. Computation is performed on the data and results are returned to c0 and stored to memory. Since loads often miss in the cache, and since stores usually issue before the store data is available, cluster 0 is specially designed to support access/execute decoupling [Smi82, EV96].

Figure 5-16 shows an overview of the load and store resources in cluster 0 and the store-data cluster. To achieve access/execute decoupling, cluster 0 must run ahead in the program stream to fetch data for the compute clusters. An important consequence is that it must continue execution after a load misses in the cache. Loads in c0 are handled as long-latency operations (Section 5.4.2). Loads fetch data into the *load data queue* (LDQ), and each transport-op indicates whether its data comes from the LDQ or the regular transport data queue. This allows c0 to continue executing new instructions after issuing a load, and it even continues to issue new loads after a load misses in the cache. As a result, the difference between c0's LDQ and a regular transport data queue is that data may be written into the queue out of order. Scale's LDQ has 8 entries.

When c0 executes a load instruction, it generates an address and allocates a slot at the tail of the LDQ. The allocation bumps a tail pointer for the queue, but the data slot itself is marked as invalid. The access is sent to the memory system along with the index of the allocated LDQ slot. Each cycle, the memory system responds with data for up to one request. The response includes the original LDQ index that was sent with the request, and the data is written into this slot and marked as valid. The responses from the memory system may return out of order with respect to the requests, but the LDQ acts as a *re-order queue*. Regardless of the order in which data is written into the LDQ, it sends out data in the same FIFO order as the original allocation. To do this it simply waits for the head entry to become valid, then it propagates this data and bumps the head pointer.

Store instructions pose a significant challenge to access/execute decoupling. Loads and compute operations can execute on different clusters to aid with decoupling. But to keep VP loads and stores coherent when they touch the same memory locations, they are both processed by cluster 0. As described previously, stores are actually split into two pieces (Section 5.2.1). The address portion of a store executes on c0, but the data portion executes on the store-data cluster (sd). The store-data cluster is not visible to the programmer, except that store instructions on c0 use special store-data registers which actually reside in sd. Any writes to the store-data registers generate writeback-ops on sd instead of c0.

When a store-address compute-op executes on c0, the generated address is written into a *store address queue* (SAQ). This leaves c0 free to continue executing other instructions, but it must stall subsequent loads if they are for the same address as a buffered store. To perform the memory disambiguation, Scale uses a conservative but energy-efficient scheme. It holds some signature bits for each buffered store in a special structure called the load/store disambiguation match queue. Since the loads and stores from different VPs do not have ordering constraints, the disambiguation bits can be taken from the VP number as well as the address. The corresponding signature for each load is checked in the match queue, and as long as any bit differs (in every valid entry) the access is guaranteed to be independent.

The SAQ in Scale holds up to 4 addresses, and through experimentation we found it sufficient to use only the low-order 2 bits of the VP number (and no address bits) in the load/store disambiguation match queue. Assuming an AIB with one or more VP loads and one VP store, 4 VPs at a time can have buffered stores in c0, but the 5th VP will have to wait for the first VP's store to complete. Eliminating address bits from the disambiguation makes it easier to stall loads early in the cycle. Otherwise, the clock period constraints might make it necessary to speculatively issue a load but nullify it if a conflict is detected late in the cycle.

The store-data cluster executes special micro-op bundles that include a writeback-op for the store-data registers and a compute-op that corresponds to the data portion of the original store instructions. As in a regular cluster, a store-data compute-op will wait until any previous writeback-ops which target its source register have resolved. Store-data compute-ops also stall until an address is available in the SAQ. When a store-data compute-op eventually executes, it reads the source data from the register file (or from the store-data chain register), pops an address from the SAQ, and issues the store to the memory system.

In addition to the store-data registers, the store-data cluster also holds the store-data predicate registers. When a store instruction is predicated based on c0/sdp, the compute-op on sd nullifies the store if the predicate evaluates to false. The store-address compute-op on c0 is not affected by the predication.

We also considered an alternative design which does not use an independent store-data cluster. Instead, cluster 0 holds a single store-data chain register, and both writes to this register and store operations are deferred in a decoupled store queue. This is a workable solution, but it means that vector-store commands interfere with both the source cluster which holds the data and cluster 0 which sends the data to the memory system. By adding private store-data registers held in an independent store-data cluster, vector-stores become simpler and more efficient. The vector store data execute directives operate solely in sd and do not interfere with c0 or the other clusters.

5.4.6 Atomic Memory Operations

Atomic memory instructions (`lw.atomic.and`, `lw.atomic.or`, and `lw.atomic.add`) can be used to synchronize VPs and allow them to share memory coherently. In Scale, an atomic VP instruction is broken into three separate cluster 0 operations: a load compute-op which locks the memory location, an arithmetic/logical compute-op, and a store compute-op which unlocks the memory location. As usual, the store is partitioned into an address and a data portion. The lock implementation is coarse-grained in Scale—the memory system actually blocks other requesters from accessing an entire cache bank until the unlock releases it.

When cluster 0 encounter a `lw.lock` compute-op, it stalls it in the decode stage until all previous stores have completed and the store-data cluster is caught up and ready to execute the corresponding store. Once it issues the load, a local c0 writeback-op waits for the data to return, possibly after a cache miss. After the data returns, the subsequent compute-op performs the atomic operation. The store address and data compute-ops issue as usual with no special logic. The memory system automatically unlocks the locked bank when it sees the store. An atomic memory operation occupies a cache bank for 7 cycles in Scale.

An atomic memory instruction may be conditional based on the VP's predicate register in c0, and the corresponding compute-ops on c0 will execute subject to c0/p as usual. However, since stores are usually only predicated based on c0/sdp, special care must be taken for the store-data compute-op. When the predicate is false, the lw.lock in c0 asserts a special signal that instructs the store-data cluster to nullify the store.

5.4.7 Cross-VP Start/Stop Queue

The software interface for a VT machine connects VPs in a unidirectional ring. Instructions can directly use operands from prevVP and write results to nextVP. The cross-VP start/stop queue is an architecturally visible structure which connects the last VP in the vector back to VP0. The control processor can also push and pop data from this queue.

Recall that each cluster in Scale has an independent cross-VP network, an overview is shown in Figure 5-17. The nextVP and prevVP ports of sibling clusters in adjacent lanes are connected, and lane 3 connects back to lane 0. As described above (Section 5.4.1), each cluster has a cross-VP queue (xvpq) and the data transfers are orchestrated with send and receive control signals.

The VTU also contains a cross-VP start/stop queue unit (xvpssq) for each cluster. The xvpssq controls the prevVP port of lane 0 in order to provide data for VP0. When lane 0 is executing micro-ops for VP0, the xvpssq sends data from its internal queue, otherwise it propagates the nextVP data

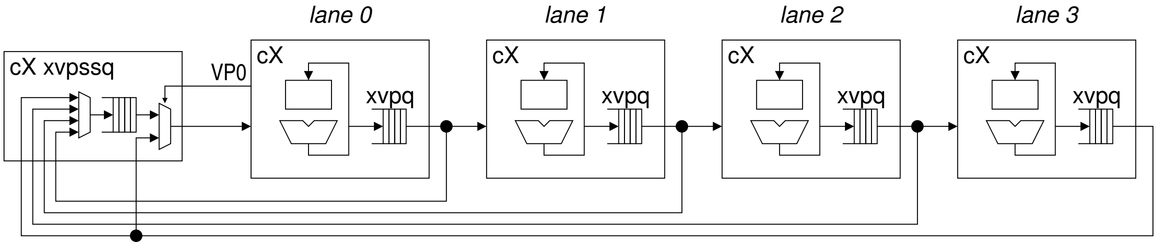


Figure 5-17: Cross-VP Start/Stop Queue. The cross-VP network for one cluster is shown (cX), all four cluster have identical networks.

directly from lane 3.

Managing the write into the start/stop queue from the last VP is more difficult since the lane which holds the last VP is a function of the configurable vector length. Furthermore, the lanes and clusters in the VTU are decoupled, so they do not necessarily agree on the vector length at any given point in time. Still, when the vector length does change, the former last VP must be allowed to complete any nextVP sends to the start/stop queue before the new last VP is allowed to write to the queue. To accomplish this, each xvpssq keeps track of its view of the current lastVP-lane, and it will only receive cross-VP data from the cluster in that lane. The clusters track their own lastVP-lane status independently. When a cluster with lastVP-lane status processes a vector length command (in the form of a config ED), it stalls until (1) its pipeline and cross-VP queue do not have buffered nextVP sends, and (2) the xvpssq agrees that it has lastVP-lane status. Then the cluster communicates with the xvpssq to pass the lastVP-lane status to another lane. This token-passing protocol handles the corner case in which back-to-back vector length commands cause a cluster to gain and lose lastVP-lane status before the previous lastVP-lane cluster has relinquished control.

When a cluster with lastVP-lane status actually executes micro-ops for the last VP, it stalls them in the decode stage until its cross-VP queue is empty. Then, any nextVP sends are sent to the cross-VP start/stop queue instead of being sent out on the regular nextVP port. Physically, a single nextVP data bus is broadcast to both the sibling cluster in the next lane and to the xvpssq. The xvpssq receives nextVP inputs from the four lanes, and a mux selects write data from the lastVP-lane. Regular send/receive synchronization between the clusters and the xvpssq ensures that a transfer only completes when the xvpssq and the cluster agree on the lastVP-lane. Although regular cross-VP transfers between VPs can have a zero-cycle latency (Section 5.4.1), a transfer through the cross-VP start/stop queue has a latency of one cycle.

A reasonable question is if the storage for the architecturally visible cross-VP start/stop queue can actually be implemented using the regular cluster cross-VP queues. Such a design would still need to carefully handle vector length changes to track which lane supplies prevVP data for VP0. However, the real problem with such an approach is the buffering required if sends to nextVP are allowed to precede receives from prevVP in an AIB. In this case each lane must be able to buffer the maximum amount of cross-VP data per AIB *plus* the cross-VP start/stop queue data. Compared to doubling the size of each cluster's cross-VP queue, it is more efficient to implement an independent start/stop queue.

5.5 Vector Memory Units

Scale's vector memory units include a vector load unit, a vector store unit, and a vector refill unit. These units process vector load and store commands and manage the interface between the VTU

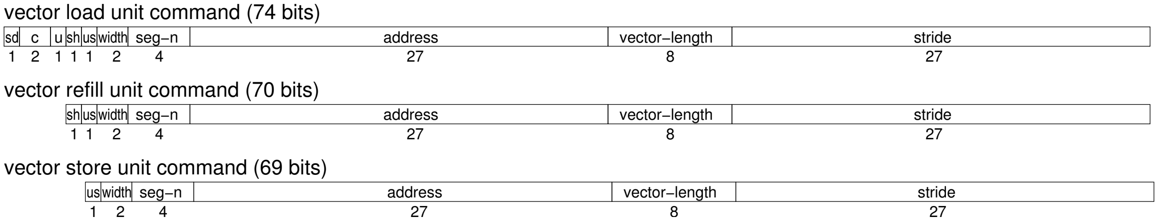

vector load unit command (74 bits)
sd c u sh us width seg−n address vector−length stride
1 2 1 1 1 2 4 27 8 27
vector refill unit command (70 bits)
sh us width seg−n address vector−length stride
1 1 2 4 27 8 27
vector store unit command (69 bits)
us width seg−n address vector−length stride
1 2 4 27 8 27

Figure 5-18: Vector memory unit commands.

and the memory system. Commands at the head of the unified VTU command queue (Figure 5-9) are reformatted into the unit commands shown in Figure 5-18. The vector memory unit commands are buffered in queues to support decoupling, in Scale the VRU command queue has 2 entries, the VLU command queue has 4 entries, and the VSU command queue has 10 entries.

The VLU and VSU each process one vector command at a time using either a single unit-stride address generator or per-lane segment-strided address generators. The VLU manages per-lane vector load data queues (VLDQs) which buffer load data and broadcast it to the clusters. The data is received in order by vector load writebacks in the destination cluster. In Scale, the VLDQs have 8 entries each. The VSU receives vector store data from the store-data cluster in each lane, and it combines the lane data into store requests for the memory system. The VRU scans vector-load commands and issues cache-line refill requests to prefetch data.

The per-lane portions of the VLU and VSU are organized together as a lane vector memory unit. The LVMU internally arbitrates between the segment-strided vector load and store requests as well as the VP load and store requests from the lane itself. It chooses up to one lane request per cycle to send to the memory system, and the cache address and data paths are shared by segment-strided accesses and VP accesses. The priority order for the LVMU in Scale is (from high to low): VP load, VP store, vector load, vector store.

An overview of the vector load and store units is shown in Figure 5-19 and the vector refill unit is shown in Figure 5-21. Together, the vector memory units have 11 address generators and 7 request ports to the memory system.

5.5.1 Unit-Stride Vector-Loads

Unit-stride is a very common memory access pattern which can readily be exploited to improve efficiency. A single cache access can fetch several data elements, thereby reducing address bandwidth demands and avoiding the bank conflicts which independent requests could incur.

For unit-stride commands, the VLU issues cache requests for up to 4 data elements at a time. Each cache request is for a naturally aligned 128-bit, 64-bit, or 32-bit data block, depending on the byte width of the vector-load (word, halfword, or byte). Although the address alignment may cause the first access to be for fewer elements, all of the subsequent accesses until the last will be for 4 elements. With each request, the VLU specifies one of 4 possible rotations for returning the 4 elements in the block to the 4 lanes, and it also specifies which of the lanes should receive the data. The VLU address generator iterates through the command until the requested number of elements equals the vector length.

The centralized VLU logic issues requests for unit-stride vector load commands, but the data itself is returned to the lanes. Since the vector load data queue in each lane is used for both unit-stride and segment-strided data, it is challenging to ensure that the data is sent to the lane in the correct order. The VLDQ is a re-order queue similar to the LDQ in cluster 0. When the VLU issues

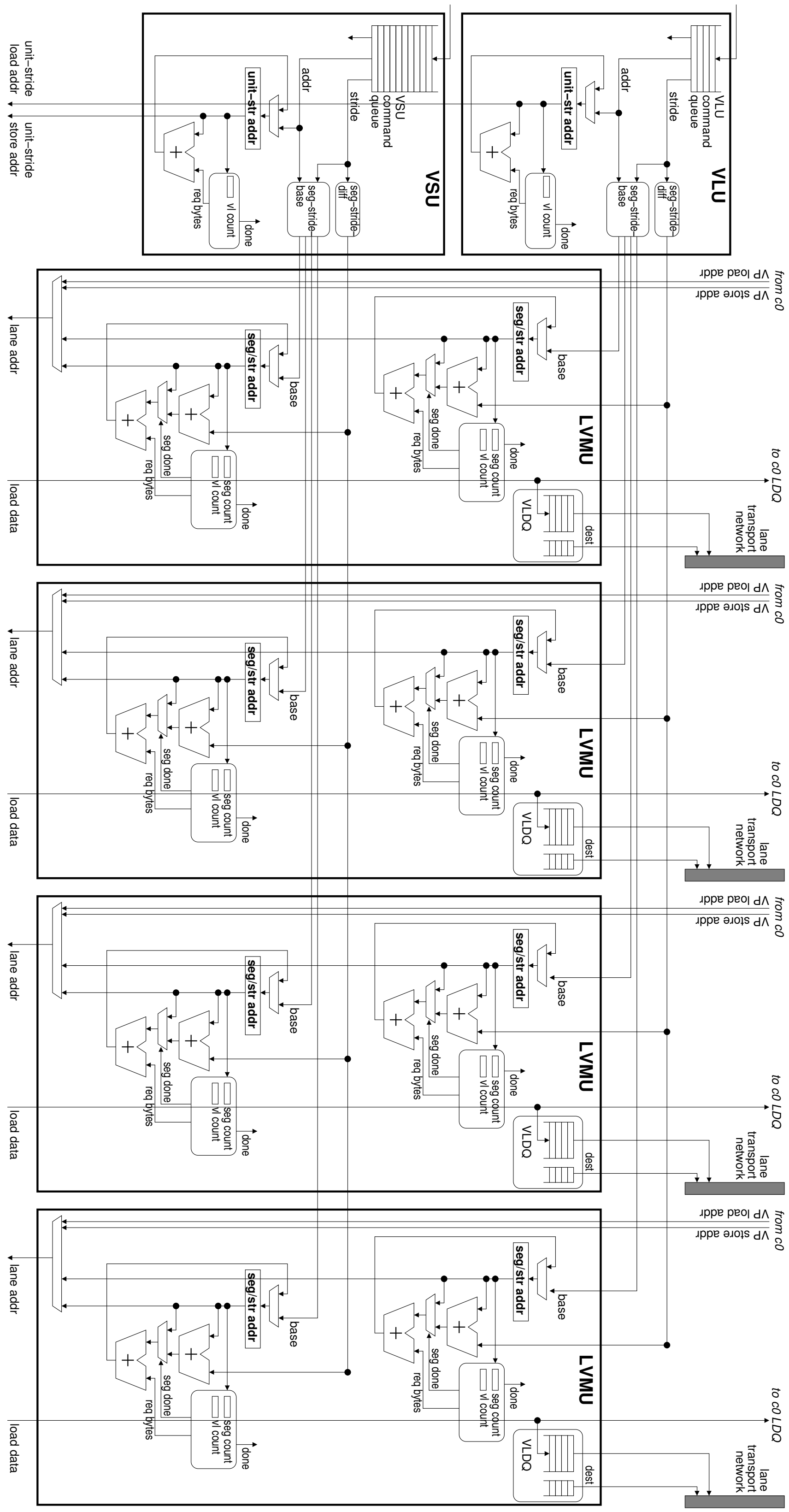


Figure 5-19: Vector load and store units microarchitecture. The major address and data paths are shown.

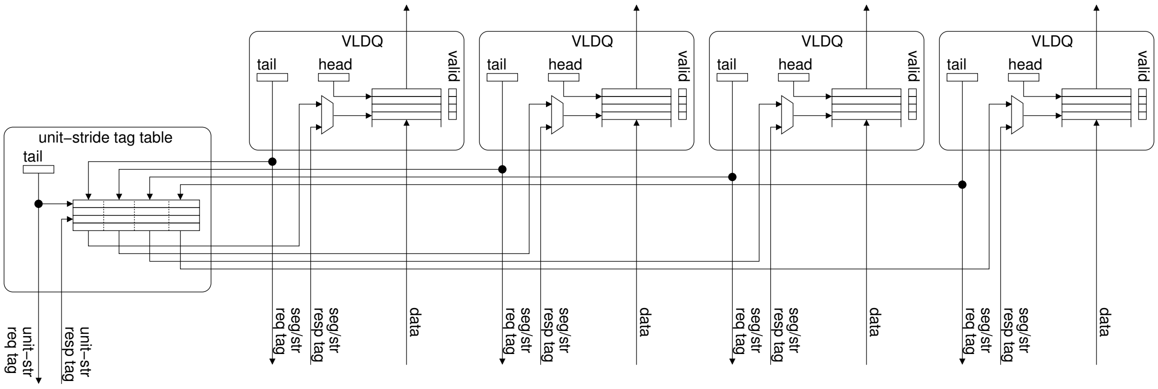


Figure 5-20: Unit-stride tag table. This is a logical view of the unit-stride tag table, physically it is distributed in the LVMUs so that the VLDQ indices do not need to be communicated back-and-forth.

a unit-stride load request, it allocates a VLDQ entry in each lane which will receive a data element from the access. The lane VLDQ indices may all be different, so the VLU stores them in a slot which it allocates in its unit-stride tag table (Figure 5-20). It sends this single tag table index as the tag associated with the memory system request. When the memory system eventually responds, the unit-stride tag table is read to determine which VLDQ slots the incoming data should be written to. Physically, the unit-stride tag table slots are distributed across the lanes so that the VLDQ indices do not need to be communicated back-and-forth to a centralized controller. In Scale, the unit-stride tag table has 4 entries, allowing up to 4 unit-stride vector-load requests at a time.

The data communication from the LVMU to the clusters in a lane fits within the regular transport/writeback framework. The LVMU has a transport-op queue which is similar to that in the clusters. Whenever a VLDQ slot is allocated for a memory request, a corresponding transport-op is queued to indicate the destination cluster. Once the transport-op reaches the head of the queue, it waits for data to be available at the head of the VLDQ. The LVMU broadcasts the data over a bus to all of the clusters, and it asserts a send signal to identify the destination cluster. The destination cluster will have a corresponding writeback-op generated by a vector load writeback execute directive, and the send and receive signals synchronize to complete the transfer.

Shared vector loads are also handled by the VLU's unit-stride address generator. It issues a request for one element at a time, and the same data is returned to all of the lanes.

5.5.2 Segment-Strided Vector-Loads

Like unit-stride memory access patterns, segment-strided access patterns present an opportunity to fetch several data elements with a single cache access. A lane's segment data is buffered in the memory system on the cache side of the read crossbar, and returned to the lane at a rate of one element per cycle. Compared to using multiple strided accesses, segments reduce address bandwidth and avoid bank conflicts.

Segment and strided vector loads are distributed to the independent lane vector memory units to be processed at a rate of up to 4 accesses per cycle. The parameters for the LVMU address generator include the segment size and the stride between segments. Regular strided loads are simply handled as single-element segments.

When it distributes a segment-strided command to the lanes, the central VLU logic must calculate a base address for each lane. Since VPs are striped across the lanes, these are calculated as the original base address plus the stride multiplied by the lane number. The address calculations do not require any multipliers: lane 0 uses the original base address, lane 1 uses the base address plus the stride, lane 2 uses the base address plus twice the stride (a shift by 1), and lane 3 uses the base address plus the stride plus twice the stride (which is computed as the lane 1 base plus twice the stride). To aid the LVMUs, the central VLU logic also calculates seg-stride-diff: the stride multiplied by the number of lanes, minus the segment size in bytes. After reaching the end of a segment, the LVMUs increment their address by this value to get to the beginning of the next segment.

Segments contain up to 15 sequential elements which may be arbitrarily aligned relative to cache blocks. The LVMU issues memory system requests for up to 4 sequential elements at a time, subject to the constraint that all the elements must be within the same naturally aligned 128-bit data block. For each access, the LVMU calculates the number of elements in its request as a function of the current address offset, and it increments its address by the requested number of bytes. The LVMU maintains a count of the number of elements requested, and once it reaches the end of a segment it increments its address by seg-stride-diff to advance to the next segment. It repeats this process to request a segment of elements for every VP according to the vector length.

To handle segment requests, the vector load data queue in the LVMU is a special re-order queue that can allocate 1–4 slots at a time. To simplify the logic, Scale only allows a segment request to issue when at least 4 slots are free. Even though several elements are requested at the same time, the memory system only returns one element per cycle and these are written into the VLDQ as usual. The transport-op for a segment request includes the requested number of elements, and it executes over several cycles to transport the data to the destination cluster.

After it issues a segment command to the LVMUs, the central VLU controller waits until they have all finished processing it before proceeding with the next vector-load command. Thus, the unit-stride and segment-strided address generators in the VLU are never active at the same time.

5.5.3 Unit-Stride Vector-Stores

Like unit-stride vector-loads, unit-stride vector-stores allow many data elements to be combined into a single cache access. The VSU breaks unit-stride stores into accesses that fit in a 128-bit block and have up to one element per lane. The data is produced by the vector store data execute directives in the lane store-data clusters, and the VSU waits until all of the active lanes are ready. The lanes' regular control and data paths are used to send the store to the memory system, with each lane providing 32-bits of data, 4 byte-enables, and a 2-bit index to indicate which word in the 128-bit block is targeted. For each active lane, the VSU calculates the address offset for the associated element within the block. The lane byte-enables and word index are a function of the byte width of the vector-store and the address offset. The data and byte-enables from all the active lanes are merged together in the memory system, and the VSU provides the address for the 128-bit store request. The VSU address generator iterates through the store command until the number of elements stored equals the vector length.

5.5.4 Segment-Strided Vector-Stores

The command management and addressing for segment and strided vector stores work similarly to vector loads. The central VSU logic calculates the base address for each lane and seg-stride-diff for the lane stride. Each LVMU address generator operates independently and breaks segments into 128-bit store requests. Regular strided stores are handled as single-element segments.

The LVMUs use the lanes' regular 32-bit data paths to send segment store data to the memory system. To take advantage of segments and reduce cache accesses, each lane manages a 128-bit

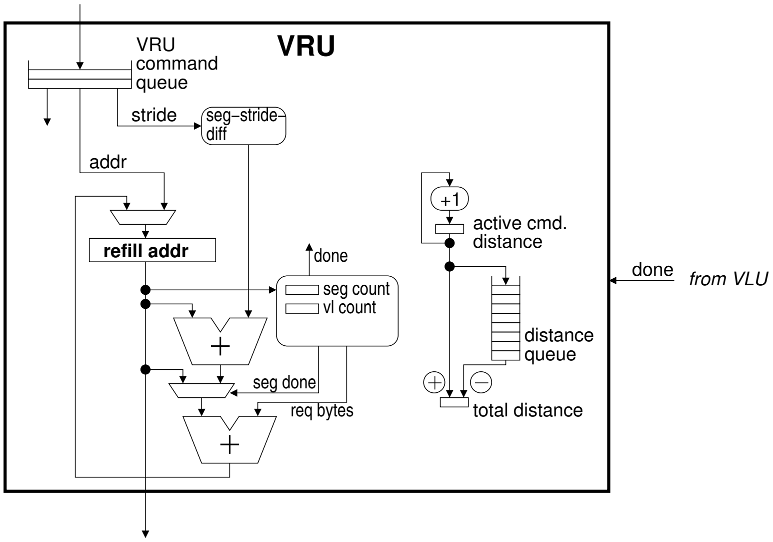


Figure 5-21: Vector refill unit.

store buffer located in the memory system on the cache side of the crossbar. The store-data cluster supplies the data and the LVMU controls which bytes in the buffer each element is written to. In addition to buffering the data, the memory system also buffers the associated byte-enables. After it reaches either the last element in a segment or the end of a 16-byte block, the LVMU issues the store request to the memory system. Once a segment is complete the LVMU increments its address by seg-stride-diff to advance to the next segment, and it repeats this process to store a segment of elements for every VP according to the vector length.

5.5.5 Refill/Access Decoupling

The vector refill unit exploits structured memory access patterns to optimize cache refills. The VRU receives the same commands as the vector load unit, and it runs ahead of the VLU issuing refill requests for the associated cache lines. The goal of refill/access decoupling is to prefetch data ahead of time so that the VLU accesses are hits. When lines are prefetched with a single VRU request, the cache does not need to buffer many VLU misses for each in-flight line.

An overview of the VRU is shown in Figure 5-21. It issues up to one refill request per cycle to the cache and has two refill modes: unit-stride and strided. It uses the unit-stride refill mode for unit-stride vector loads and segment-strided vector loads with positive strides less than the cache line size. In this mode, the VRU calculates the number of cache lines in the memory footprint once when it begins processing a command, and then issues cache-line refill requests to cover the entire region. The hardware cost for the segment-strided conversion includes a comparator to determine the short stride and a small multiplier to calculate the vector length (7 bits) times the stride (5 bits). For the strided refill mode, the VRU repeatedly issues a refill request to the cache and increments its address by the stride until it covers the entire vector. It also issues an additional request whenever a segment straddles a cache-line boundary.

Three important throttling techniques prevent the VLU and the VRU from interfering with each other. (1) Ideally, the VRU runs far enough ahead so that all of the VLU accesses hit in the cache. However, when the program is memory bandwidth limited, the VLU can always catch up with the VRU. The VLU interference wastes cache bandwidth and miss resources and can reduce overall

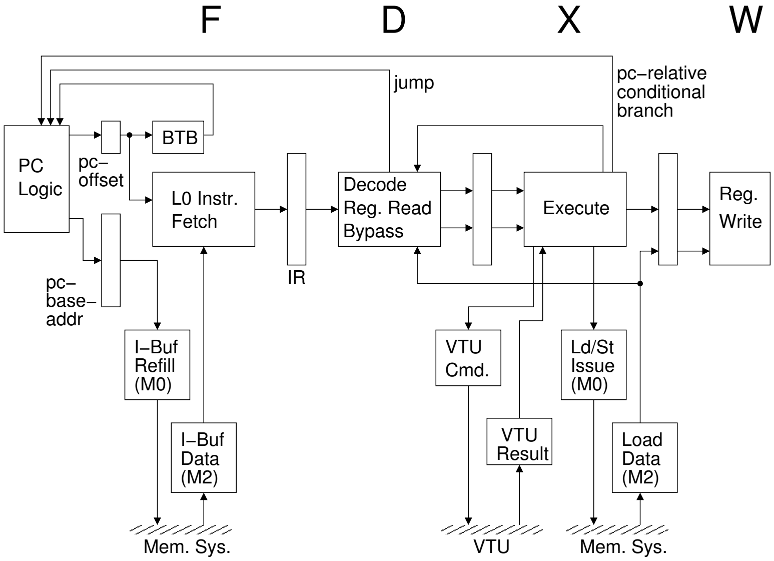


Figure 5-22: Control processor pipeline overview. The CP pipeline includes fetch (F), decode (D), execute (X), and writeback (W) stages. A decoupled fetch unit manages the program counter and a small L0 instruction buffer. Load data writebacks are decoupled from the execution pipeline and use a second register file write port. The control processor issues commands to the VTU command queue, and it may also writeback results from the VTU.

performance. As a solution, the VLU is throttled to prevent it from having more than a small number of outstanding misses. (2) The VLU has four address generators and can therefore process segment-strided vector loads that hit in the cache at a faster rate than the VRU. Whenever the VLU finishes a vector load command before the VRU, the VRU aborts the command. (3) If the VRU is not constrained, it will run ahead and use all of the miss resources, causing the cache to block and preventing other requesters from issuing cache hits. The VRU is throttled to reserve resources for the other requesters.

When a program is compute-limited, the VRU may run arbitrarily far ahead and evict data that it has prefetched into the cache before the VLU even accesses it. To avoid this, the VRU is also throttled based on the *distance* that it is running ahead of the VLU. To track distance, the VRU counts the number of cache-line requests that it makes for each vector load command (including hits). After finishing each command, it increments its total distance count and enqueues that command's distance count in its distance queue (see Figure 5-21). Whenever the VLU finishes a vector load, the VRU pops the head of the DistQ and decrements its total distance count by the count for that command. A single distance count is sufficient for programs that use the cache uniformly, but it can fail when a program has many accesses to a single set in the cache. As a solution, the VRU can maintain per-set counts, and throttle based on the maximum per-set distance. To reduce the hardware cost, the distance-sets that the VRU divides its counts into may be more coarse-grained than the actual cache sets.

5.6 Control processor

The control processor ISA is a simplified version of MIPS with no branch delay slot. Scale's control processor design is based on a traditional single-issue RISC pipeline. Regular instructions go through fetch (F), decode (D), execute (X), and writeback (W) stages. Load and store instructions issue to the memory system during X. Figure 5-22 shows an overview of the control processor pipeline.

To interface with Scale's non-blocking cache, the control processor includes a decoupled load data writeback unit. The CP may have multiple loads in flight, and these may return from the memory system out of order. The load data writeback unit tracks the destination register indices for the in-flight loads and uses a dedicated register file port to writeback the data. Subsequent instructions stall if they depend on an outstanding load, but otherwise they are free to execute and writeback before a previous load completes. Load data returning from the cache may be bypassed directly to a dependent instruction in stage D of the pipeline. Scale's cache has a two cycle hit latency, so there will be two bubble cycles between a load and use. To cover the pipelining and cache hit latency, Scale's control processor supports up to 3 in-flight loads.

The CP processes VTU commands as described in Section 5.3.1. It issues commands to the VTU command queue during stage X of the pipeline. After issuing a command that returns a result (`vprd` or `xvppop`), the control processor stalls until the data is available from the VTU. After issuing a `vsync` or `vpsync` command, a subsequent load or store instruction will stall until the VTU synchronization is complete (Section 5.3.1).

The control processor supports user and kernel execution modes. The 32-bit virtual addresses in Scale programs are simply truncated to index physical memory. However, to aid debugging, user mode addresses are restricted to the top half of virtual memory (i.e., they have bit-31 set) and violations trigger an exception [Asa98]. Scale's exception and interrupt handling interface is derived from the MIPS system control coprocessor (CP0) [KH92]. Exceptions include address protection and misalignment faults, `syscall` and `break` instructions, illegal instructions, and arithmetic overflow. Scale also traps floating point instructions and emulates them in software. Scale supports both external and timer-based interrupts. The control processor handles exceptions and interrupts by switching to kernel mode and jumping to a predefined entry point in the kernel instruction space.

5.6.1 L0 Instruction Buffer

The control processor fetch unit uses a small L0 instruction buffer which is located on the cache side of the read crossbar. This buffer simplifies the CP pipeline by providing a single-cycle access latency and avoiding cache arbitration. Scale's L0 buffer caches 128 bytes, and it can hold any contiguous sequence of 32 instructions whose starting address is aligned on a 16-byte boundary. By not restricting the alignment to 128-byte boundaries, the L0 buffer is able to cache short loops regardless of their starting address.

The CP's program counter and the tag state for the L0 buffer are intricately linked. A 28-bit *pc-base-addr* indicates the starting address of the L0 sequence (which may be any naturally aligned 16-byte block), and a 5-bit *pc-offset* points to the current instruction within the sequence. By default, the 5-bit pc-offset increments every cycle to implement sequential instruction fetch. Whenever it wraps around (from 31 to 0), the pc-base-addr is incremented to point to the next 128-byte block. Unconditional jumps resolve in stage D of the pipeline and always load both PC registers. Conditional branches resolve in stage X of the pipeline, but they do not update pc-base-addr if the target is located within the L0 buffer. A short loop may initially incur two L0 misses if the pc-offset wraps around the first time it executes. However, after the backward branch the L0 buffer will be loaded based on the starting address of the loop.

Whenever pc-base-addr changes, the fetch unit automatically issues eight 16-byte requests to the primary cache to refill the L0 buffer. This causes a 2-cycle penalty to retrieve the first new block (assuming that it is present in the primary cache). The CP fetch unit maintains valid bits for the

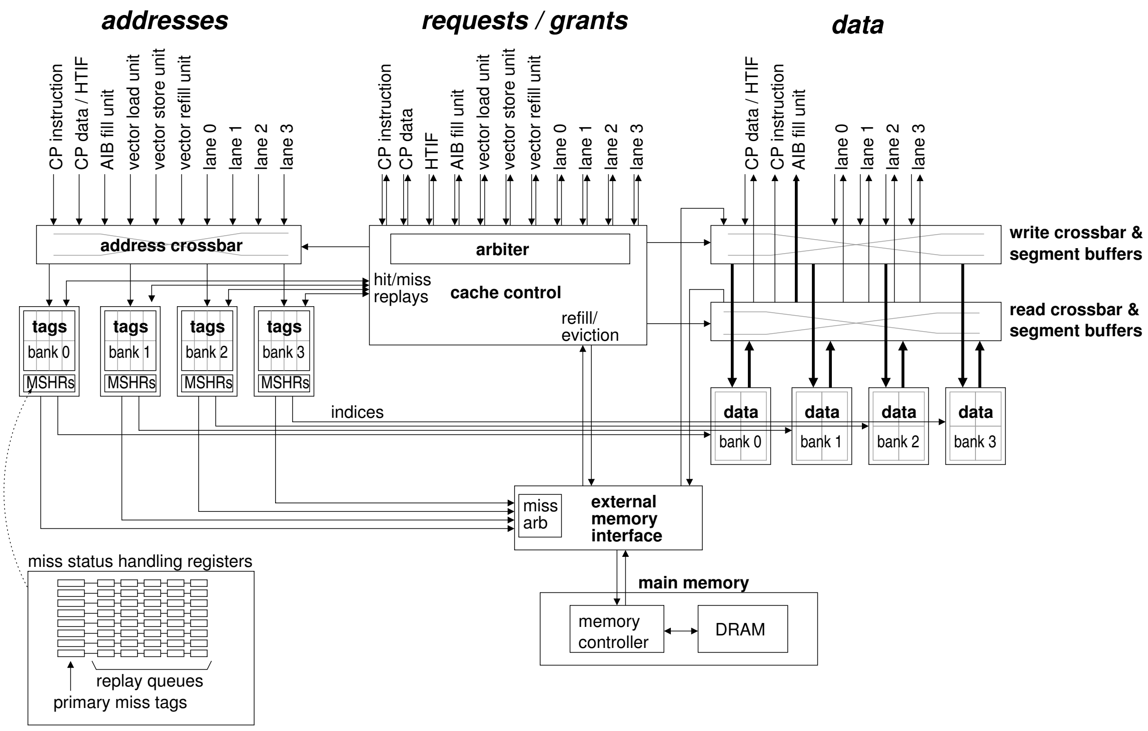


Figure 5-23: Cache microarchitecture.

blocks within the L0 buffer, and it sets these when its refill accesses hit in the primary cache. The fetch unit can generate several cache misses when it tries to refill the L0 buffer. Instead of tracking these, whenever the block it is trying to access is not valid, it simply polls the primary cache until it gets a hit.

With regular sequential instruction fetch, a taken branch (in X) must nullify the invalid instructions in stages F and D of the pipeline. Scale uses a small branch target buffer (BTB) to eliminate the penalty for taken branches within the L0 instruction buffer. Scale's BTB has 4 entries which each include 5-bit source and target pc-offsets. Every cycle the current pc-offset is broadcast to all the entries in the BTB. If there is no match, the fetch unit uses sequential instruction fetch. If there is a match, a taken branch is predicted and the BTB provides the target pc-offset. Entries are inserted into the BTB whenever a branch within the L0 buffer is taken, and invalidated whenever a branch within the L0 buffer is not taken. All entries are invalidated whenever the L0 buffer moves to a new block of instructions.

5.7 Memory System

The units on the processor side of the Scale VT architecture issue load and store requests to the memory system. These include 8-, 16-, 32-, and 128-bit accesses, as well as segment requests. Each cycle, requests to the memory system are either granted or denied. A store is complete once accepted by the memory system. Loads use a request/response protocol that allows variable latency and re-ordering of requests. When requesters support multiple outstanding loads, each is tagged with a unique identifier that the memory system tracks and returns with the response.

The memory system is an architectural abstraction that may have various implementations. The

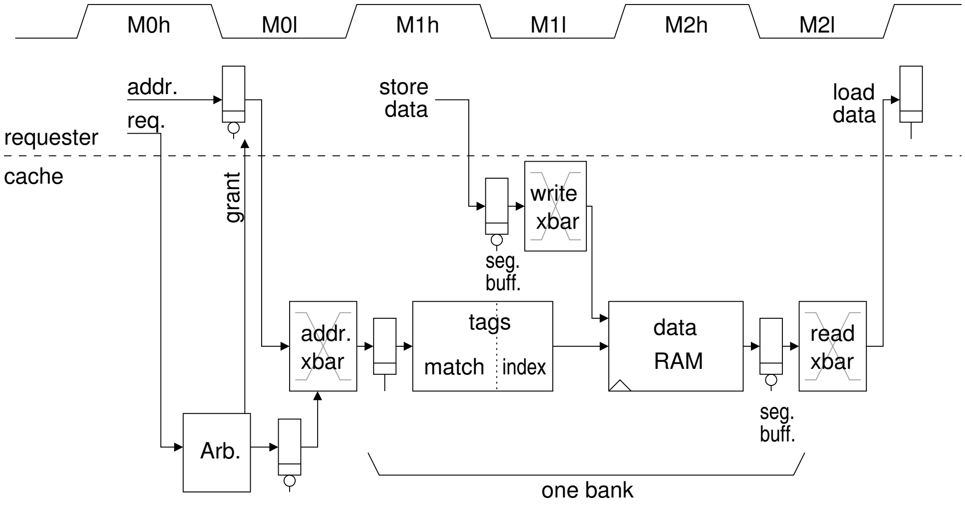


Figure 5-24: Cache access pipeline.

Scale processor includes an on-chip 32 KB non-blocking primary cache and a connection to an external main memory, an overview is shown in Figure 5-23. The cache uses a line size of 32 bytes, and it uses CAM tags to provide 32-way set-associativity. To provide sufficient bandwidth for the many requesters, the cache is partitioned into 4 independent banks. Each bank can read or write 128-bits every cycle, and cache lines are striped across the banks. An arbiter controls access to the banks. Read and write crossbars provide the interface between the bank and requester data ports, and the crossbars include load and store segment buffers.

Cache accesses use a three-stage pipeline shown in Figure 5-24. Requests are sent at the beginning of M0, and the arbiter determines grant signals by the end of the same cycle. Up to 4 successful requests are selected by the address crossbar during M0 and extending into M1. The tags are searched in M1 to determine whether each access is a hit or a miss and to obtain the RAM indices. Also during M1 the write crossbar prepares the store data. The data bank is read or written during M2, and load data is returned across the read crossbar.

The control processor load/store unit and the host interface share a port into the cache.

5.7.1 Non-blocking Cache Misses

The non-blocking Scale cache continues to operate after a miss. The cache supports both *primary misses* and *secondary misses*. A primary miss is the first miss to a cache line and causes a refill request to be sent to main memory, while secondary misses are accesses which miss in the cache but are for the same cache line as an earlier primary miss. Misses are tracked in the miss status handling registers using *primary miss tags* and *replay queues*. Primary miss tags hold the address of an in-flight refill request and are searched on every miss to determine if it is a secondary miss. A primary miss allocates a new primary miss tag to hold the address, issues a refill request, and adds a new replay queue entry to a linked list associated with the primary miss tag. Replay queue entries contain information about the access including the corresponding cache line offset, the byte width, the destination register for loads, and a pending data pointer for stores. A secondary miss adds a new entry to the appropriate replay queue, but does not send an additional refill request to main memory. The bank blocks if it runs out of primary miss tags or replay queue entries. The Scale cache supports 32 primary misses (8 per bank), and these may each have up to 5 pending replays.

When a primary miss occurs, a new destination line for the refill is allocated in FIFO order. If the victim line is dirty, an eviction is needed. Refill requests are buffered in a memory request queue in each bank, and the external memory interface unit arbitrates between the banks. It sends refill requests to the external main memory controller, followed by the eviction request if necessary. When a refill returns from main memory, the arbiter blocks other requesters to allow the line to be written into the data bank. Then the cache controller processes the replay queue cycle-by-cycle to return load data to the requesters and to update the line with the pending store data. Since multiple banks can be returning load replay data to the same requester on a cycle, replays must arbitrate for the writeback ports through the main cache arbiter.

Data for pending store misses is held in a per-bank *pending store data buffer* and store replay queue entries contain pointers into this buffer. The storage for Scale's pending store data buffer is implemented as 8 extra 128-bit rows in each cache bank data array (0.5 KB total). Secondary store misses are merged into an already allocated store data buffer entry where possible.

We also experimented with a no-write-allocate policy. In such a design, stores misses are held in the replay queues while waiting to be sent to memory. These non-allocating stores are converted to allocating stores if there is a secondary load miss to the same cache line before the data is sent out. A no-write-allocate policy is supported by the Scale microarchitectural simulator, but it is not implemented in the Scale chip.

5.7.2 Cache Arbitration

The cache arbitrates between 10 requesters and also handles replays, refills, and evictions. It uses a fixed priority arbiter for each of the four banks, and each request targets only one bank as determined by the low-order address bits. The internal cache operations are given priority over the external requesters: refills have highest priority since the external memory interface can not be stalled, followed by evictions, and then replays. Of the requesters, the host interface is given highest priority, followed by the control processor data and instruction requests, and then the VTU instruction fetch. These requesters are given high priority to ensure that they can make progress even if all the lanes are continuously accessing a single bank. The lanes themselves are next, and the arbiter uses a round-robin policy between them to ensure fairness. The vector memory units have lowest priority: first the vector load unit, then the vector store unit, and finally the vector refill unit. Given the relatively high cache access bandwidth and the decoupling resources in the VTU, we found that the ordering of requester priorities does not usually have a large effect on performance.

In addition to arbitrating for the cache bank, some requesters must arbitrate for other shared resources. The vector load unit and the lanes both return data to the lane writeback ports, so a VLU request is killed whenever there is a lane request, even if they are for different banks. Also, since load replays use the same writeback ports as cache hits, they preclude any original request by the same unit. This also means that VLU replays kill lane requests in order to reserve the lane writeback ports. Finally, a host interface request kills a CP data access since they use the same address and data paths to access the cache.

5.7.3 Cache Address Path

The Scale cache has 10 requester address ports. The addresses are held in a low-latch by the requesters. The address crossbar consists of four ten-input muxes to select the active address in each bank. The outputs of the address crossbar go through a high-latch and are sent to the tag banks.

Each tag bank is organized as eight 32-entry fully-associative subbanks, with low-order address bits selecting the subbank. An address is broadcast to all the entries in the subbank, and these each

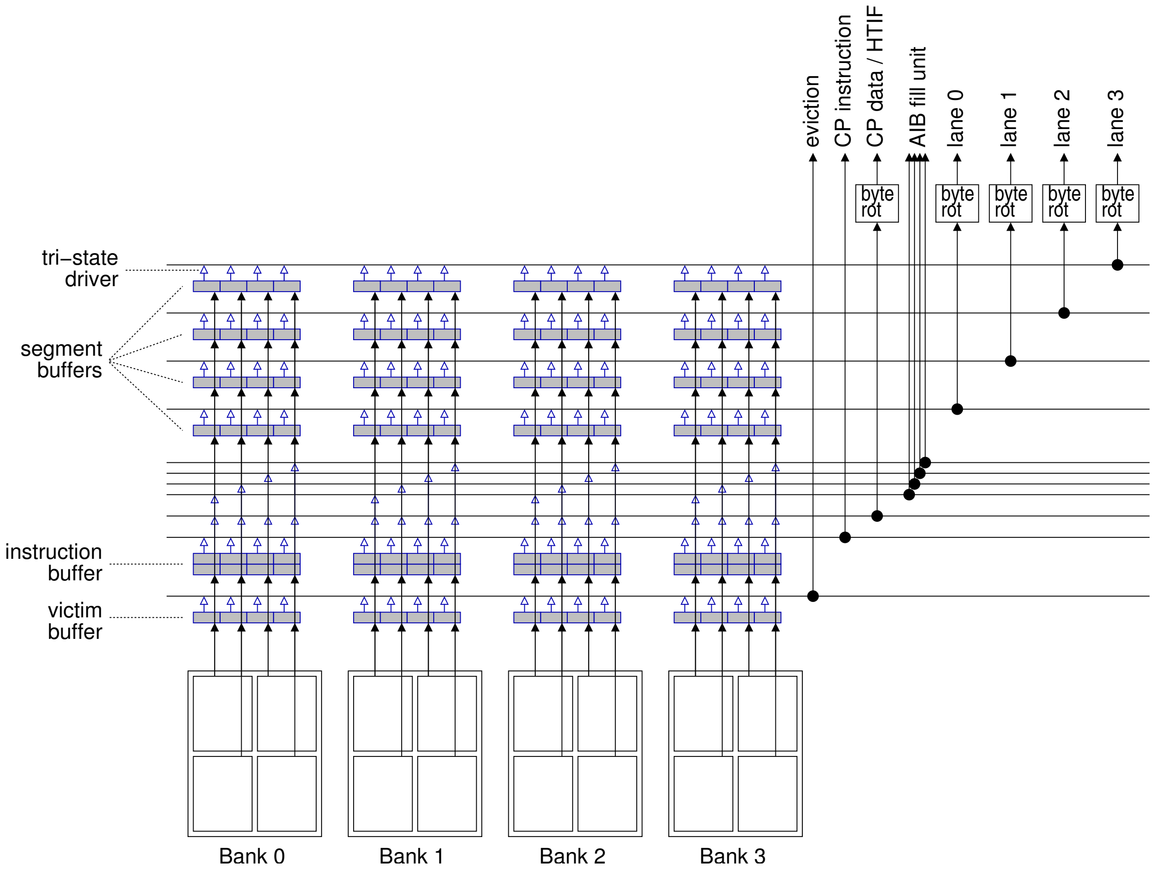


Figure 5-25: Read crossbar. Each data bank is divided into four 32-bit-wide subbanks. The horizontal busses are driven by tri-state drivers. All wires in the figure are 32-bits wide.

have a local comparator. The 32 match lines are OR-ed together to produce the hit signal, and also encoded into a 5-bit set index. These 5 bits are combined with 4 low-order address bits to create a 9-bit index to address a 128-bit word in the data bank.

5.7.4 Cache Data Path

Each data bank provides a 128-bit read/write port and includes support for per-byte write-enables. The banks are partitioned into four 32-bit subbanks, and only a subset of these are activated when an access is for fewer than 128-bits. The data bank connects to read and write crossbars, shown in Figures 5-25 and 5-26, to send data to and receive data from the requesters.

All of the requester read ports are 32-bits, except for the 128-bit VTU instruction port. The lane ports are either controlled together by a unit-stride vector-load request from the VLU, or they are controlled by independent load requests from the lanes. Each 32-bit requester port is driven by one of 16 possible 32-bit words from the data banks (there are four banks with four output words each). The 16-to-1 multiplexers are implemented as long busses with distributed tri-state drivers. Byte and halfword load data for the lanes and the CP is constructed in two stages: first the correct 32-bit word is broadcast on the requester read port, then muxes on this port control the byte selection and sign-extension.

In some cases the output data bus from a bank directly drives the requester busses in the read

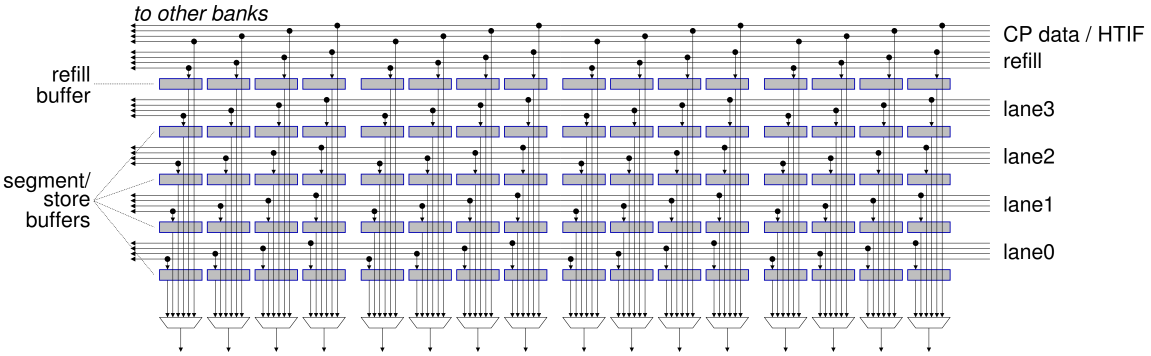


Figure 5-26: Write crossbar. Only one bank is shown, but the horizontal wires extend to all four banks and the crossbar in each is identical. All wires in the figure are 8-bits wide. The 16 crossbar muxes send 128-bits of write data to the bank.

crossbar. This is how the cache controls the VTU instruction port and the control processor data port. Other requester ports include one or more 128-bit buffers adjacent to the bank. This is the case for the lane segment buffers (1 slot), the control processor instruction buffer (2 slots), and the victim eviction buffer (2 slots). The buffers are implemented as low-latches that the bank data passes through before connecting to the requester read busses. The transparent latches allow the data buffers to be both written and read on the same cycle that the data bank is accessed.

The write crossbar receives 32-bit store data from six requesters: the four lanes, the combined control processor and host interface data port, and the external memory interface unit. This data is broadcast to all of the banks across long busses. For unit-stride vector-store requests, the data from multiple lanes is merged together and sent to a single bank's write port. Lane stores are handled independently and may be sent to different banks. The requesters are responsible for aligning byte and halfword data within their 32-bit port, which they do by simply replicating bytes four times and halfwords twice. Therefore, each byte in a bank's 128-bit write port comes from the corresponding byte in one of the requester ports, and a simple 6-to-1 multiplexer selects the data.

The write crossbar includes a 128-bit store buffer for each lane adjacent to each bank. Similar to the segment load buffers, these are implemented with transparent low-latches, allowing the lane store data to pass through within the M1 pipeline stage. The latches are used to implement the lanes' segment store buffers, and they are also used to temporarily buffer data when a store request is denied by the cache arbiter.

5.7.5 Segment Accesses

When the cache receives a segment load request from a lane, a data block is read from the bank and the first element is sent over the read crossbar as usual. The data is also latched in the lane's 128-bit segment buffer adjacent to the bank. The remaining elements in the segment are sent over the read crossbar one-by-one during the following cycles. The cache arbiter continues to reserve the lane writeback port during these cycles, but the bank itself is available to handle other requests. The original request tag is returned with the first element and the tag is incremented with each subsequent element (the lane uses the tag to determine which VLDQ slot to write the data to). If a segment load misses in the cache, it will eventually be replayed using exactly the same control logic as the hit case.

Unlike segment loads, the element accesses for segment stores are handled by the lanes. Each lane manages its segment buffer and builds up a request over several cycles, including both the data and the byte-enables. The store request is sent to the cache as the last element is written to the buffer, and it is handled as a regular single-cycle 128-bit access with no special logic.

Even though each lane has only one logical load segment buffer and one logical store segment buffer, the lane buffers are physically replicated in each bank. This allows the wide 128-bit datapaths between the buffers and the data RAMs to remain within the bank, while narrow 32-bit datapaths in the crossbar link the lanes to the buffers. The area impact of the replication is not great since the buffers are physically located beneath the crossbar wires.

Chapter 6

Scale VT Processor Implementation

This chapter describes a prototype hardware implementation of the Scale vector-thread architecture. The Scale VT processor has 7.14 million transistors, around the same number as the Intel Pentium II, the IBM/Motorola PowerPC 750, or the MIPS R14000. However, as part of a university research project, we could not devote industry-level design effort to the chip building task. In this chapter, we describe the implementation and verification flow that enabled us to build an area and power efficient prototype with only two person-years of effort. The Scale chip proves the plausibility of vector-thread architectures and demonstrates the performance, power, and area efficiency of the design.

6.1 Chip Implementation

The Scale design contains about 1.4 million gates, a number around 60 times greater than a simple RISC processor. To implement Scale with limited manpower, we leverage Artisan standard cells and RAM blocks in an ASIC-style flow targeting TSMC's 180 nm 6 metal layer process technology (CL018G). We use Synopsys Design Compiler for synthesis and Cadence SoC Encounter for place-and-route.

In establishing a tool flow to build the Scale chip, we strove to make the process as automated as possible. A common design approach is to freeze blocks at the RTL level and then push them through to lower-levels of implementation using hand-tweaked optimizations along the way. Instead, our automated flow enables an iterative chip-building approach more similar to software compilation. After a change in the top-level RTL, we simply run a new iteration of synthesis and place-and-route to produce an updated full chip layout, even including power and ground routing. Despite this level of automation, the tool flow still allows us to pre-place datapath cells in regular arrays, to incorporate optimized memories and custom circuit blocks, and to easily and flexibly provide a custom floorplan for the major blocks in the design.

With our iterative design flow, we repeatedly build the chip and optimize the source RTL or floorplan based on the results. We decided to use a single-pass flat tool flow in order to avoid the complexities involved in creating and instantiating hierarchical design blocks. As the design grew, the end-to-end chip build time approached two days. We are satisfied with this methodology, but the runtimes would become prohibitive for a design much larger than Scale.

6.1.1 RTL Development

Our early research studies on Scale included the development of a detailed microarchitecture-level simulator written in C++. This simulator is flexible and parameterizable, allowing us to easily evaluate the performance impact of many design decisions. The C++ simulator is also modular with a design hierarchy that we carried over to our hardware implementation.

We use a hybrid C++/Verilog simulation approach for the Scale RTL. After implementing the RTL for a block of the design, we use Tenison VTOC to translate the Verilog into a C++ module with input and output ports and a clock-tick evaluation method. We then wrap this module with the necessary glue logic to connect it to the C++ microarchitectural simulator. Using this methodology we are able to avoid constructing custom Verilog test harnesses to drive each block as we develop the RTL. Instead, we leverage our existing set of test programs as well as our software infrastructure for easily compiling and running directed test programs. This design approach allowed us to progressively expand the RTL code base from the starting point of a single cluster, to a single lane, to four lanes; and then to add the AIB fill unit, the vector memory unit, the control processor, and the memory system. We did not reach the milestone of having an initial all-RTL model capable of running test programs until about 14 months into the hardware implementation effort, 5 months before tapeout. The hybrid C++/Verilog approach was crucial in allowing us to easily test and debug the RTL throughout the development.

6.1.2 Datapath Pre-Placement

Manually organizing cells in bit-sliced datapaths improves timing, area, and routing efficiency compared to automated placement [CK02]. However, to maintain an iterative design flow, a manual approach must still easily accommodate changes in the RTL or chip floorplan. We used a C++-based procedural datapath tiler which manipulates standard cells and creates design databases using the OpenAccess libraries. The tool allows us to write code to instantiate and place standard cells in a virtual grid and create nets to connect them together. After constructing a datapath, we export a Verilog netlist together with a DEF file with relative placement information.

We incorporate datapath pre-placement into our CAD tool flow by separating out the datapath modules in the source RTL; for example, the cluster datapaths for Scale include the ALU, shifter, and many 32-bit muxes and latches. We then write tiler code to construct these datapaths and generate cell netlists. During synthesis we provide these netlists in place of the source RTL for the datapath modules, and we flag the pre-placed cells as `dont_touch`. In this way, Design Compiler can correctly optimize logic which interfaces with the datapath blocks. During the floorplanning step before place-and-route, we use scripts to flexibly position each datapath wherever we want on the chip. These scripts process the relative placement information in the datapath DEF files, combining these into a unified DEF file with absolute placement locations. We again use `dont_touch` to prevent Encounter from modifying the datapaths cells during placement and optimization. We use Encounter to do the datapath routing automatically; this avoids the additional effort of routing by hand, and we have found that the tool does a reasonable job after the datapath arrays have been pre-placed.

As a simple example of the ease with which we can create pre-placed datapath arrays, Figure 6-1(a) shows a small snippet of Verilog RTL from Scale which connects a 32-bit mux with a 32-bit latch. Figure 6-1(b) shows the corresponding C++ code which creates the pre-placed datapath diagrammed in Figure 6-1(c). The placement code is simple and very similar to the RTL, the only extra information is the output drive strength of each component. The supporting component builder libraries (`dpMux2` and `dpLatch_h_en`) each add a column of cells to the virtual grid in the tiler (`tl`). By default, the components are placed from left to right. In this example, the `dpMux2` builder creates each two-input multiplexer using three NAND gates. The component builders also add the necessary clock gating and driver cells on top of the datapath, and the code automatically sets the size of these based on the bit-width of the datapath.

```
wire [31:0] xw_sd_pn;
wire [31:0] w_sd_np;

dpMux2 #(32)
  xw_sd_mux
    ( .s    (xw_sdsel_pn),
      .i0   (xw_reg_pn),
      .i1   (xw_sdc_pn),
      .o    (xw_sd_pn));

dpLatch_h_en #(32)
  w_sd_hl
    ( .clk  (clk),
      .d_n  (xw_sd_pn),
      .en_p (x_sden_np),
      .q_np (w_sd_np));
```

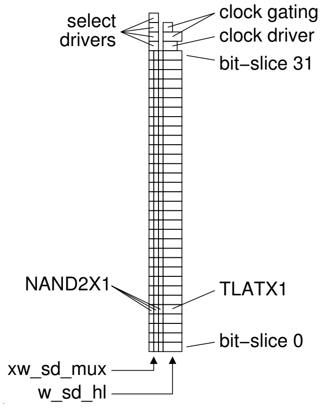
(a) Verilog RTL

```
Tiler tl;
tl.addNet  ( "xw_sd_pn", 32 );
tl.addNet  ( "w_sd_np", 32 );

dpMux2 ( tl, Drive::X1, 32,
    "xw_sd_mux",
    "xw_sdsel_pn",
    "xw_reg_pn",
    "xw_sdc_pn",
    "xw_sd_pn");

dpLatch_h_en ( tl, Drive::X1, 32,
    "w_sd_hl",
    "clk",
    "xw_sd_pn",
    "x_sden_np",
    "w_sd_np");
```

(b) C++ pre-placement code



(c) Datapath cells

Figure 6-1: Datapath pre-placement code example.

We used our datapath pre-placement infrastructure to create parameterizable builders for components like muxes, latches, queues, adders, and shifters. It is relatively straightforward to assemble these components into datapaths, and easy to modify the datapaths as necessary. In the end, we pre-placed 230 thousand cells, 58% of all standard cells in the Scale chip.

6.1.3 Memory Arrays

Wherever possible, we use single-port and two-port (read/write) RAMs created by the Artisan memory generators. These optimized memories provide high-density layout and routing with six or eight transistors per bitcell. To incorporate the RAM blocks into our CAD tool flow, we use behavioral RTL models for simulation. These are replaced by the appropriate RAM data files during synthesis and place-and-route.

Artisan does not provide memory generators suitable for Scale's register files and CAM arrays. We considered using custom circuit design for these, but decided to reduce design effort and risk by building these components out of Artisan standard cells. The two-read-port two-write-port register file bitcell is constructed by attaching a mux cell to the input of a latch cell with two tri-state output ports. We found that the read ports could effectively drive 8 similar bitcells. Since our register file has 32 entries, we made the read ports hierarchical by adding a second level of tri-state drivers. For the CAM bitcell we use a latch cell combined with an XOR gate for the match. The OR reduction for the match comparison is implemented using a two-input NAND/NOR tree which is compacted into a narrow column next to the multi-bit CAM entry. For the cache-tag CAMs which have a read port in addition to the match port, we add a tri-state buffer to the bitcell and use hierarchical bitlines similar to those in the register file.

We pay an area penalty for building the register file and CAM arrays out of standard cells; we estimate that our bitcells are 3–4× larger than a custom bitcell. However, since these arrays occupy about 17% of the core chip area, the overall area impact is limited to around 12%. We found that these arrays are generally fast enough to be off of the critical path; for example, a register file read only takes around 1.4 ns including the decoders. The power consumption should also be competitive with custom design, especially since our arrays use fully static logic instead of pre-charged dynamic bitlines and matchlines.

6.1.4 Clocking

Scale uses a traditional two-phase clocking scheme which supports both latches and flip-flops. The datapaths primarily use latches to enable time-borrowing across clock phases, as eliminating hard clock-edge boundaries makes timing optimization easier. Also, the pre-placed register file, CAM, and RAM arrays use latches to reduce area. A standard cell latch optimized for memory arrays with a tri-state output is around 41% smaller than a flip-flop. The control logic for Scale primarily uses flip-flops, as this makes the logic easier to design and reason about.

The Scale design employs aggressive clock-gating to reduce power consumption. The architecture contains many decoupled units, and typically several are idle at any single point in time. Control signals for operation liveness and stall conditions are used to determine which pipeline registers must be clocked each cycle. In many cases these enable signals can become critical paths, especially when gating the clock for low latches and negative-edge triggered flip-flops and RAMs. We make the enable condition conservative where necessary in order to avoid slowing down the clock frequency. We considered adding global clock-gating for entire clusters in Scale, but abandoned this due to the critical timing and complexity of ensuring logical correctness of the enable signal.

We incorporate clock-gating logic into the custom designed datapath components. Typically, an enable signal is latched and used to gate the clock before it is buffered and broadcast to each cell of a multi-bit flip-flop or latch. This reduces the clock load by a factor approximately equal to the width of the datapath, e.g. 32× for a 32-bit latch. We implement similar gating within each entry in the register file and CAM arrays, but we also add a second level of gating which only broadcasts the clock to all of the entries when a write operation is live.

For clock-gating within the synthesized control logic, we use Design Compiler's automatic `insert_clock_gating` command. To enable this gating, we simply make the register updates conditional based on the enable condition in the source Verilog RTL. Design Compiler aggregates common enable signals and implements the clock-gating in a similar manner to our datapath blocks. Out of 19,800 flip-flops in the synthesized control logic, 93% are gated with an average of about 11 flip-flops per gate. This gating achieves a 6.5× reduction in clock load when the chip is idle.

In addition to gating the clock, we use data gating to eliminate unnecessary toggling in combinational logic and on long buses. For example, where a single datapath latch sends data to the shifter, the adder, and the logic unit, we add combinational AND gates to the bus to only enable the active unit each cycle.

6.1.5 Clock Distribution

The clock on the Scale chip comes from a custom designed voltage-controlled oscillator that we have used successfully in previous chips. The VCO frequency is controlled by the voltage setting on an analog input pad, and it has its own power pad for improved isolation. We can also optionally select an external clock input, and divide the chosen root clock by any integer from 1–32. There are a total of around 94,000 flip-flops and latches in the design. An 11-stage clock tree is built automatically by Encounter using 688 buffers (not counting the buffers in the pre-placed datapaths and memory arrays). The delay is around 1.5–1.9 ns, the maximum trigger-edge skew is 233 ps, and the maximum transition time at a flip-flop or latch is 359 ps. We use Encounter to detect and fix any hold-time violations.

Figure 6-2 highlights all of the clocked cells in Scale and shows their relative clock skew. The regular cell layout in the pre-placed datapaths and memory arrays makes it easier for the clock tree generator to route the clock while minimizing skew.

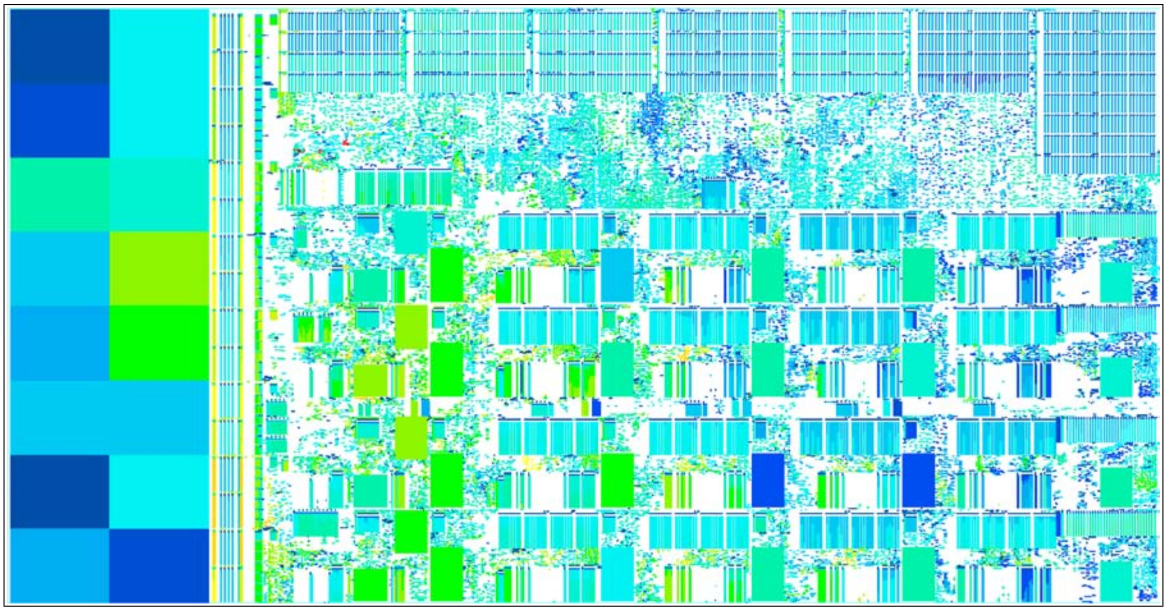


Figure 6-2: Chip clock skew. All of the clocked cells are highlighted. Cells with the lowest clock latency are dark blue and cells with the highest clock latency are red. The few number of red cells have an intentionally delayed clock edge to allow time-borrowing across pipeline stages.

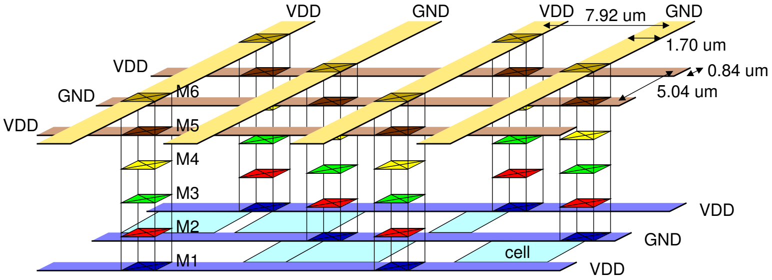


Figure 6-3: Chip power grid. A small section of two standard cell rows is shown.

6.1.6 Power Distribution

The standard cell rows in Scale have a height of nine metal-3 or metal-5 tracks. The cells get power and ground from metal-1 strips that cover two tracks, including the spacing between the tracks. The generated RAM blocks have power and ground rings on either metal-3 (for two-port memories) or metal-4 (for single-port SRAMs).

A power distribution network should supply the necessary current to transistors while limiting the voltage drop over resistive wires. We experimented with some power integrity analysis tools while designing Scale, but mainly we relied on conservative analysis and an over-designed power distribution network to avoid problems. We tried running fat power strips on the upper metal layers of the chip, but these create wide horizontal routing blockages where vias connect down to the standard cell rows. These wide blockages cause routing problems, especially when they fall within the pre-placed datapath arrays.

We settled on a fine-grained power distribution grid over the entire Scale chip, a diagram is shown in Figure 6-3. In the horizontal direction we route alternating power and ground strips on metal-5, directly over the metal-1 strips in the standard cell rows. These use two routing tracks,

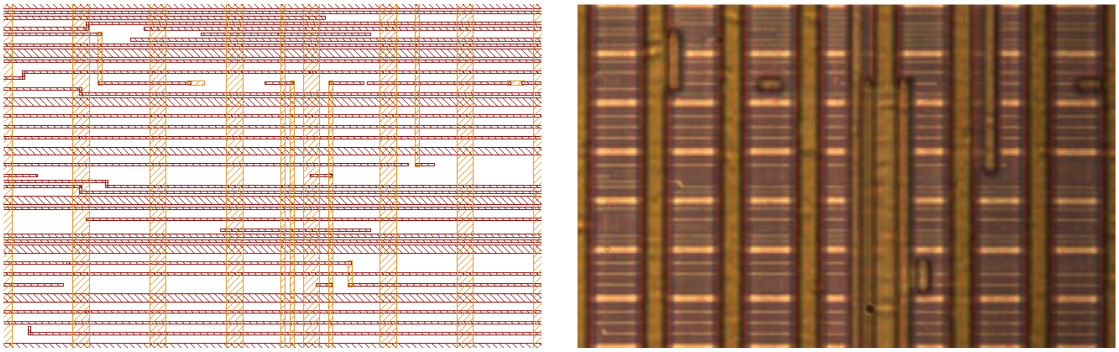


Figure 6-4: Plot and corresponding photomicrograph showing chip power grid and routing. Metal layers 5 (horizontal) and 6 (vertical) are visible, and the height is seven standard cell rows (about 35 μm).

leaving seven metal-3 and metal-5 tracks unobstructed for signal routing. In the vertical direction, we route alternating power and ground strips on metal-6. These cover three metal-2 and metal-4 tracks, leaving nine unobstructed for signal routing. Figure 6-4 shows a plot and photomicrograph of the power grid, including some data signals that are routed on metal layers 5 and 6. Figure 6-5 shows plots and photomicrographs of the lower metal layers, 1–4. Regularly spaced pillars are visible where the power grid contacts down to the metal 1 power strips in the standard cell rows, and the data signals are routed around these blockages. Overall, the power distribution uses 21% of the available metal-6 area and 17% of metal-5. We estimate that this tight power grid can provide more than enough current for Scale, and we have found that the automatic router deals well with the small blockages.

Another power supply concern is the response time of the network to fast changes in current (di/dt). At our target frequency range, the main concern is that the chip has enough on-chip decoupling capacitance to supply the necessary charge after a clock edge. Much of the decoupling capacitance comes from filler cells which are inserted into otherwise unused space on the chip, as shown in Figure 6-6. Around 20% of the standard cell area on the chip is unused, and filler cells with decoupling capacitance are able to fit into 80% of this unused area, a total of 1.99 mm^2. Scale uses a maximum of around 2.4 nC (nano-coulombs) per cycle, and we estimate that the total on-chip charge storage is around 21.6 nC. This means that, even with no response from the external power pads, a single cycle can only cause the on-chip voltage to drop by around 11%. We deemed this an acceptable margin, and we rely on the external power pads to replenish the supply before the next cycle.

6.1.7 System Interface

The Scale chip test infrastructure includes three primary components: a host computer, a general test baseboard, and a daughter card with a socket for the Scale chip (see Figure 6-7). Scale is packaged in a 121-pin ceramic package (PGA-121M). The test baseboard includes a host interface and a memory controller implemented on a Xilinx FPGA as well as 24 MB of SDRAM, adjustable power supplies with current measurement, and a tunable clock generator. The host interface uses a low-bandwidth asynchronous protocol to communicate with Scale, while the memory controller and the SDRAM use a synchronous clock that is generated by Scale. Using this test setup, the host computer is able to download and run programs on Scale while monitoring the power consumption at various voltages and frequencies.

The Scale chip has 32-bit input and output ports to connect to an external memory controller.

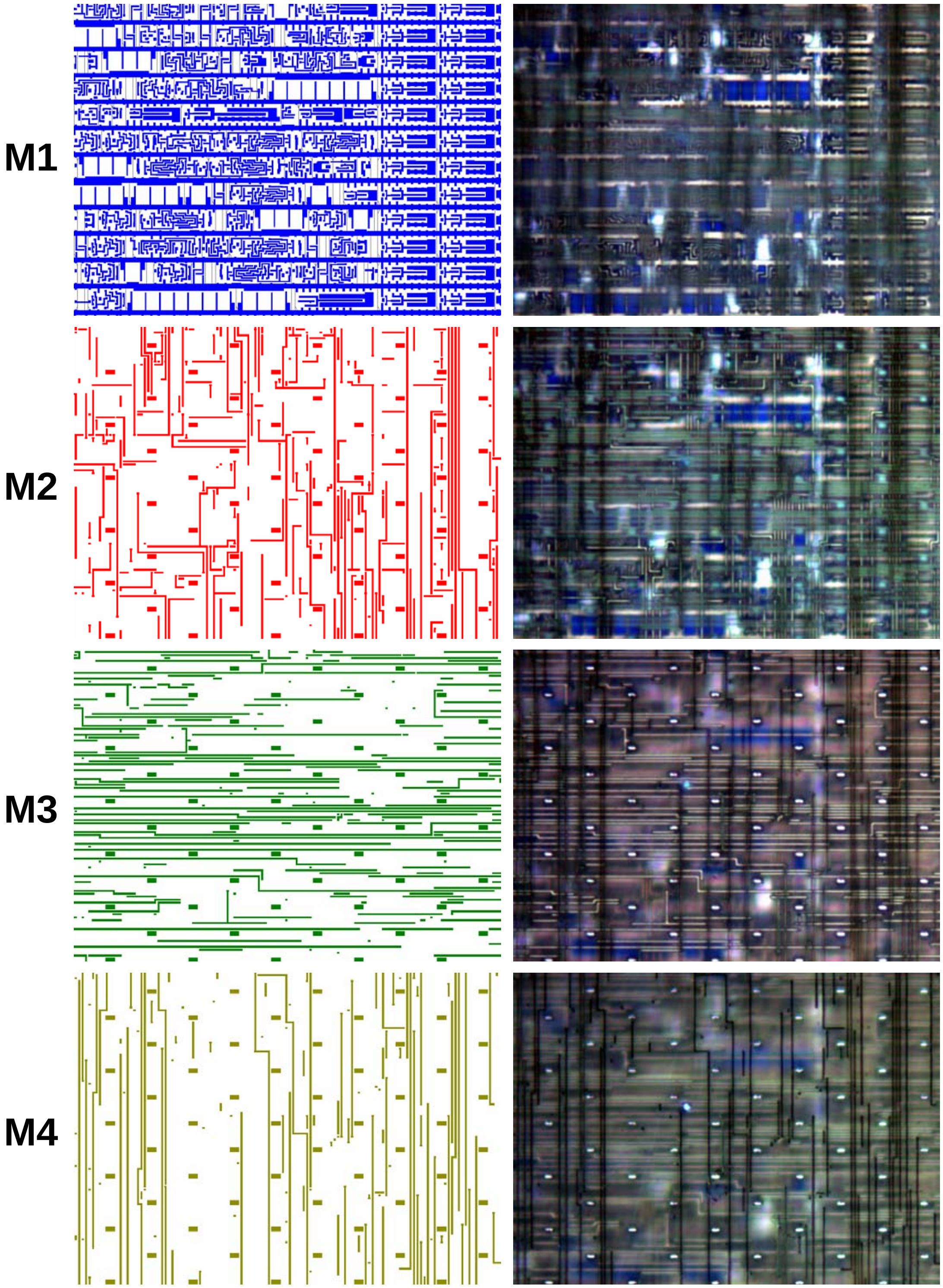


Figure 6-5: Plots and corresponding photomicrographs of chip metal layers. The photomicrographs are of the same die (which has had metal layers 5 and 6 removed), but they are taken at different focal depths. The area shown is around 80 μm×60 μm.

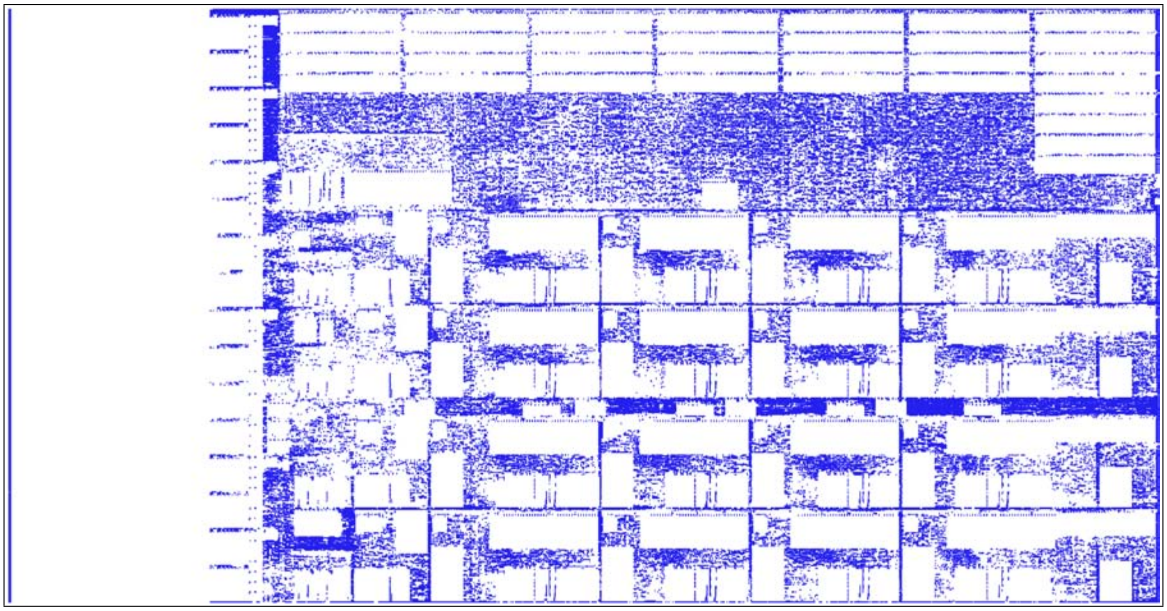


Figure 6-6: Chip filler cells. All of the decoupling capacitance filler cells in the chip are highlighted. Note that the image scaling can make the highlighted cells appear to occupy more area than they actually do.

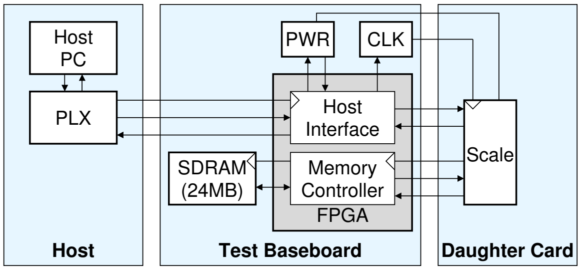


Figure 6-7: Block diagram of Scale test infrastructure.

With its non-blocking cache, the chip itself has the ability to drive a high-bandwidth memory system through these ports. However the existing test infrastructure limits the memory interface to 16-bits in each direction. More significantly, the SDRAM in the test infrastructure is only 12-bits wide and the frequency of the memory controller is limited by the outdated Xilinx FPGA. The few bits of memory bandwidth per Scale clock cycle will certainly limit performance for applications with working sets that do not fit in Scale's 32 KB cache. However, we can emulate systems with higher memory bandwidth by running Scale at a lower frequency so that the memory system becomes relatively faster. To enable this, the Scale clock generator supports configurable ratios for Scale's internal clock, the external memory system clock, and the memory interface bus frequency. The chip even supports a special DDR mode in which data is transferred on both edges of the Scale clock. This allows us to emulate a memory bandwidth of up to 4 bytes in and out per Scale cycle.

For testing purposes, the Scale chip supports a mode of operation in which cache misses are disabled and the memory system is limited to the on-chip 32 KB RAM. In this mode, the cache arbiter and data paths are still operational, but all accesses are treated as hits. As the tape-out deadline approached, it became apparent that the frequency of the design was limited by some long paths through the cache tags and the cache miss logic which we did not have time to optimize. Since the primary goal for the chip is to demonstrate the performance potential of the vector-thread unit, and given the memory system limitations imposed by the testing infrastructure, we decided to optimize timing for the on-chip RAM mode of operation. Unfortunately, this meant that we had to essentially ignore some of the cache paths during the timing optimization, and as a result the frequency when the cache is enabled is not as fast as it could be.

6.2 Chip Verification

We ensure correct operation of the Scale chip using a three-step process: First, we establish that the source RTL model is correct. Second, we prove that the chip netlist is equivalent to the source RTL. Third, we verify that the chip layout matches the netlist.

6.2.1 RTL Verification

Our overall strategy for verifying the Scale RTL model is to compare its behavior to the high-level Scale VM simulator (Section 4.11.3). We use the full chip VTOC-generated RTL model, and the C++ harness downloads programs over the chip I/O pins through the host interface block. After the program runs to completion, the output is read from memory using this same interface and compared to reference outputs from the Scale VM simulator. In addition to a suite of custom directed tests and the set of application benchmarks for Scale, we developed VTorture, a random test program generator. The challenge in generating random tests is to create legal programs that also stress different corner cases in the design. VTorture randomly generates relatively simple instruction sequences of various types, and then randomly interleaves these sequences to construct complex yet correct programs. By tuning parameters which control the breakdown of instruction sequence types, we can stress different aspects of Scale. Although our full chip RTL simulator runs at a modest rate, on the order of 100 cycles per second, we used a compute farm to simulate over a billion cycles of the chip running test programs.

In addition to running test programs on the full RTL model, we also developed a test harness to drive the Scale cache on its own. We use both directed tests and randomly generated load and store requests to trigger the many possible corner cases in the cache design.

The VTOC-based RTL simulator models the design using two-state logic. This is sufficient for most testing, however, we must be careful not to rely on uninitialized state after the chip comes out of reset. To complete the verification, we use Synopsys VCS as a four-state simulator. We did not construct a new test harness to drive VCS simulations. Instead, we use the VTOC simulations to generate value change dump (VCD) output which tracks the values on the chip I/O pins every cycle. Then we drive a VCS simulation with these inputs, and verify that the outputs are correct every cycle. Any use of uninitialized state will eventually propagate to `X`'s on the output pins, and in practice we did not find it very difficult to track down the source.

Unfortunately, the semantics of the Verilog language allow unsafe logic in which uninitialized `X` values can propagate to valid 0/1 values [Tur03]. In particular, `X`'s are interpreted as 0's in conditional `if` or `case` statements. We dealt with this by writing the RTL to carefully avoid these situations, and by inserting assertions (error messages) to ensure that important signals are not `X`'s. For additional verification that the design does not rely on uninitialized state, we run two-state VTOC simulations with the state initialized with all zeros, all ones, and random values.

6.2.2 Netlist Verification

The pre-placed datapaths on Scale are constructed by hand, and therefore error prone. We had great success using formal verification to compare the netlists from the pre-placed datapaths to

process technology	TSMC 180 nm, 6 metal layers
transistors	7.14 M
gates	1.41 M
standard cells	397 K
flip-flops and latches	94 K
core area	16.61 mm^2
chip area	23.14 mm^2
frequency at 1.8 V	260 MHz
power at 1.8 V	0.4–1.1 W
design time	19 months
design effort	24 person-months

Table 6.1: Scale chip statistics.

the corresponding datapath modules in the source RTL. The verification is done in seconds using Synopsys Formality.

As a double-check on the tools, we also use formal verification to prove equivalence of the source RTL to both the post-synthesis netlist and the final netlist after place-and-route. These Formality runs take many hours and use many gigabytes of memory, but they removed the need for gate-level simulation.

6.2.3 Layout Verification

The final verification step is to check the GDSII layout for the chip. We convert the final chip netlist from Verilog to a SPICE format and use Mentor Graphics Calibre to run a layout-versus-schematic (LVS) comparison. This check ensures that the transistors on the chip match the netlist description. LVS can catch errors such as routing problems which incorrectly short nets together. We also use Calibre to run design rule checks (DRC) and to extract a SPICE netlist from the layout.

We use Synopsys Nanosim to simulate the extracted netlist as a final verification that the chip operates correctly. This is the only step which actually tests the generated Artisan RAM blocks, so it serves as an important check on our high-level Verilog models for these blocks. The Nanosim setup uses VCD traces to drive the inputs and check the outputs every cycle (as in the VCS simulations, the VCD is generated by simulations of the source RTL). To get the I/O timing correct, these simulations actually drive the core chip module which does include the clock generator. We crafted separate Nanosim simulations to drive the chip I/O pins and verify the clock generator module. We also use Nanosim simulations to measure power consumption.

6.3 Chip Results and Measurements

We began RTL development for Scale in January of 2005, and we taped out the chip on October 15, 2006. Subtracting two months during which we implemented a separate test chip, the design time was 19 months, and we spent about 24 person-months of effort. We received chips back from fabrication on February 8, 2007, and we were able to run test programs the same day. The off-chip memory system was first operational on March 29, 2007. The first silicon has proven to be fully functional with no known bugs after extensive lab testing.

Figure 6-8 shows an annotated plot of the final Scale design and Figure 6-9 shows a single cluster. Figure 6-10 shows die photos of the fabricated chip and Table 6.1 summarizes the design statistics.

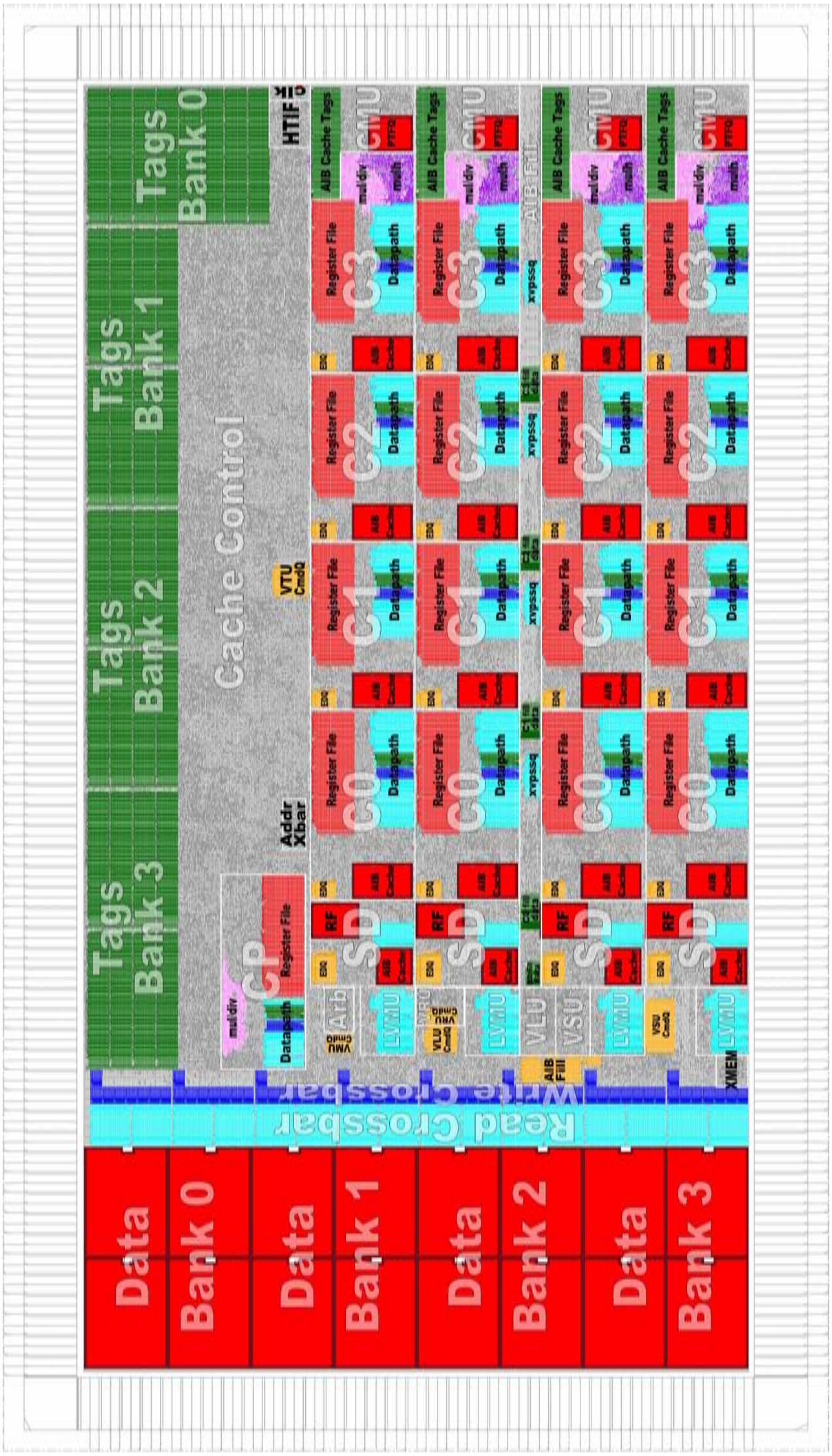


Figure 6-8: Final Scale chip plot (annotated). The labeled blocks roughly correspond to the microarchitecture diagram in Figure 5-1.

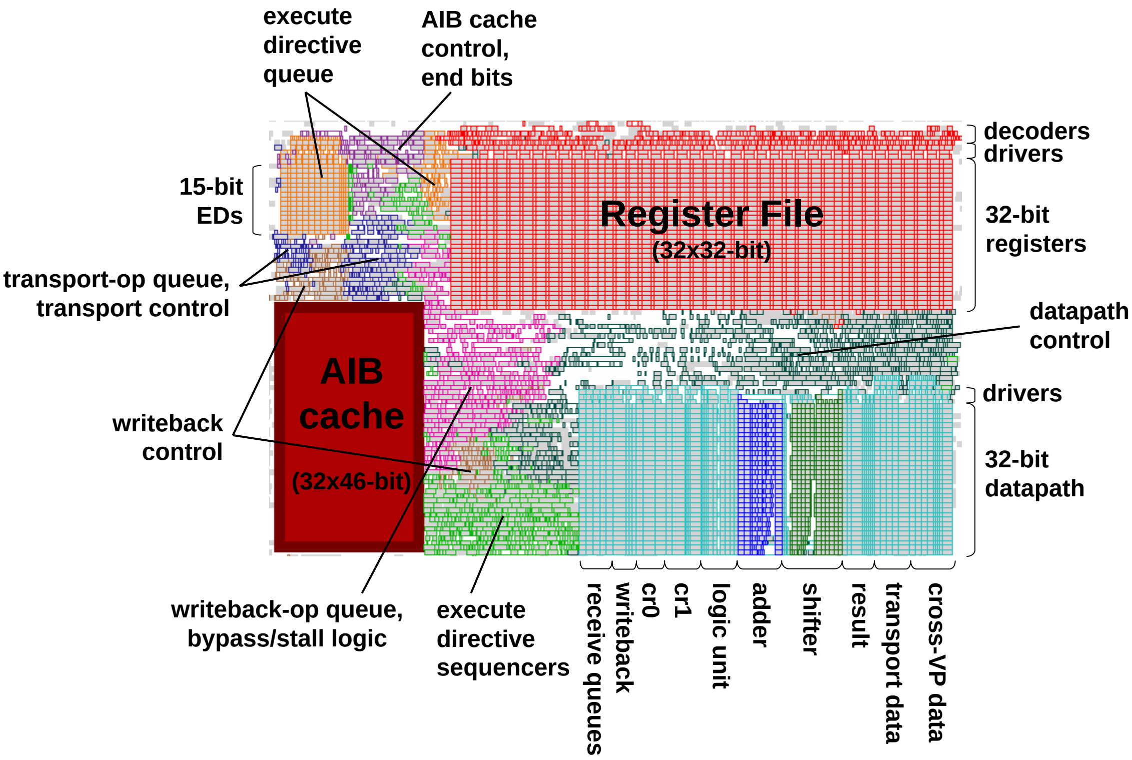


Figure 6-9: Plot of a basic cluster implementation (cluster 2). The major components are labeled, and many of these correspond to the microarchitecture diagram in Figure 5-13.

6.3.1 Area

Scale's core area is 16.61 mm^2 in 180 nm technology. At this size it is a suitable candidate for an efficient and flexible core in a processor array architecture. If implemented in 45 nm technology, Scale would only use around 1 square millimeter of silicon area, allowing hundreds of cores to fit in a single chip. Even with the control overheads of vector-threading and its non-blocking cache, Scale's 16 execution clusters provide relatively dense computation. The core area is around the same size as a single tile in the 16-tile RAW microprocessor [TKM$^+$03]. In 130 nm technology, around 40 Scale cores could fit within the area of the two-core TRIPS processor [SNG$^+$06]. In 90 nm technology, Scale would be around 28% the size of a single 14.8 mm^2 SPE from the 8-SPE Cell processor [FAD$^+$06]. These comparisons are not one-to-one since the various chips provide different compute capabilities (e.g. floating-point) and have different amounts of on-chip RAM, but Scale's compute density is at least competitive with these other many-operation-per-cycle chips.

We initially estimated that the Scale core would be around 10 mm^2 [KBH$^+$04a]. This estimate assumed custom layout for the cache tags and the register files, whereas the actual Scale chip builds these structures out of standard cells. The estimate was also based on a datapath cell library which was more optimized for area. Furthermore, the initial study was made before any RTL was written for the Scale chip, and the microarchitecture has evolved considerably since then. For example, we added the store-data cluster and refined many details in the non-blocking cache design.

Tables 6.2 and 6.3 show the area breakdown for the Scale chip and for an individual cluster. The area is split roughly evenly between the vector-thread unit and the memory system, with the control processor only occupying around 2% of the total. As somewhat technology-independent reference points, the control processor uses the gate equivalent area of around 3.3 KB of SRAM, and the

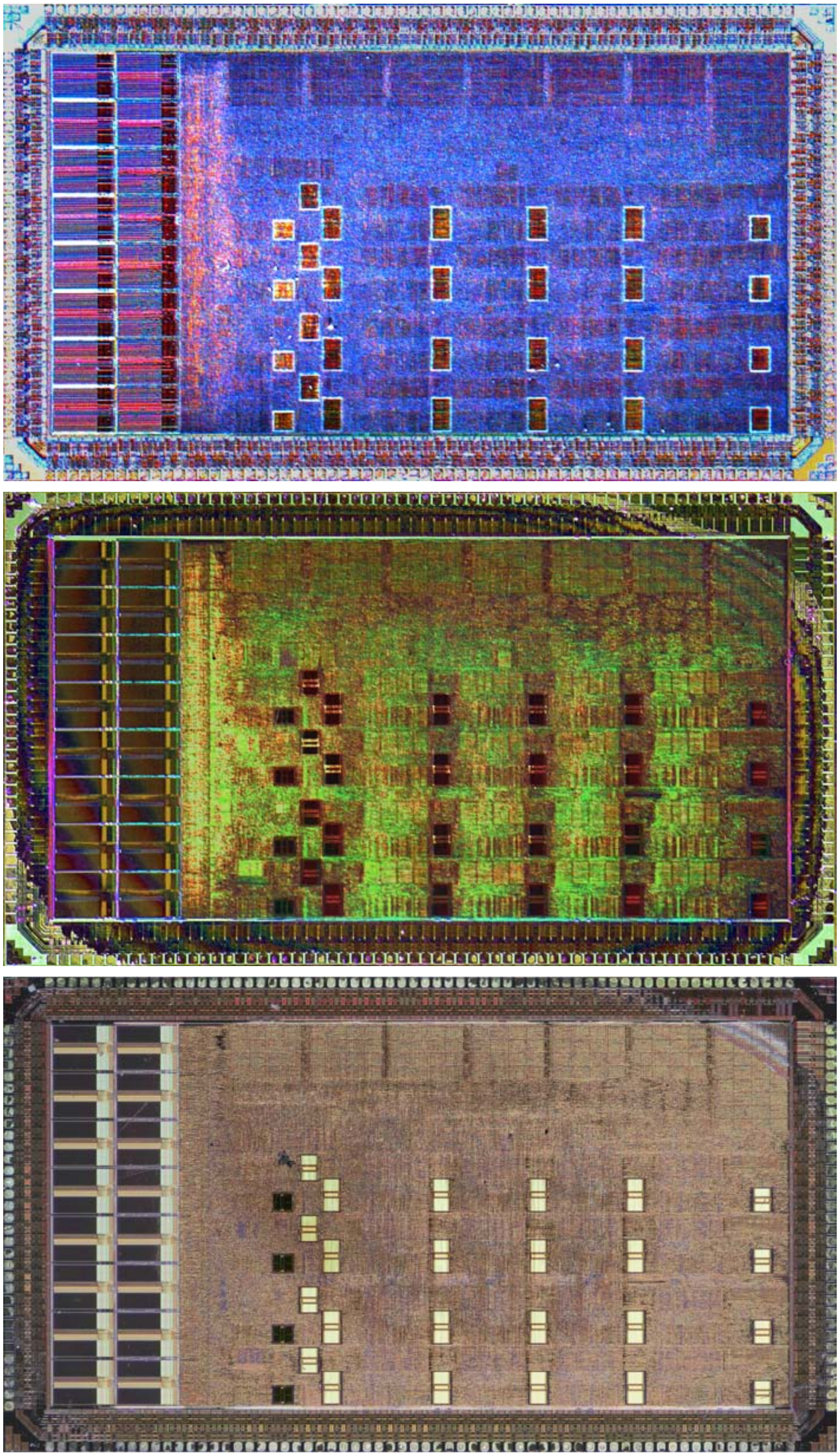


Figure 6-10: Scale die photos. These chips were processed after fabrication to remove the top metal layers. The chip in the top photomicrograph was chemically etched, and the lower two photomicrographs are of a chip that was parallel lapped. The features visible in the different photos depend on the lighting conditions.

	Gates (thousands)	Gates percentage			Cells (thousands)	RAM bits
Scale (total)	1407.0	100.0			397.0	
Control Processor	28.8	2.0	100.0		10.9	
Register File	9.8		34.0		3.0	
Datapath	4.7		16.3		1.8	
Mult/Div	2.5		8.7		1.2	
VLMAX	1.4		4.9		0.8	
Control Logic	10.3		35.8		4.1	
Vector-Thread Unit	678.2	48.2	100.0		211.4	33792
VTU command queues	11.2		1.7		4.0	
VLU	3.1		0.5		1.7	
VSU	3.0		0.4		1.5	
VRU	3.0		0.4		1.4	
AIB Fill Unit	12.8		1.9		5.3	
XVPSSQs	5.6		0.8		2.0	
Lane (4×)	159.6		23.5	100.0	48.8	8448
CMU	13.4			8.4	4.2	896
C0	31.8			19.9	10.0	1472
C1	28.5			17.9	8.5	1472
C2	28.1			17.6	8.4	1472
C3	35.6			22.3	11.4	1472
SD	13.5			8.5	2.8	1664
LVMU	6.6			4.1	2.7	
Memory System	688.7	48.9	100.0		171.2	266240
Data Arrays	287.6		41.8			266240
Tag Arrays	177.1		25.7		83.1	
Tag Logic	61.6		8.9		33.0	
MSHRs	43.5		6.3		15.2	
Arbiter	1.7		0.2		0.9	
Address Crossbar	1.1		0.2		0.6	
Read Crossbar	50.7		7.4		13.0	
Write Crossbar	27.1		3.9		11.7	
Other	38.3		5.6		13.7	
Host Interface	3.1	0.2			1.1	
Memory Interface	2.1	0.1			0.8	

Table 6.2: Scale area breakdown. Many of the blocks are demarcated in Figure 6-8. The *Gates* metric is an approximation of the number of circuit gates in each block as reported by Cadence Encounter. Different sub-breakdowns are shown in each column of the *Gates percentage* section. The *Cells* metric reports the actual number of standard cells, and the *RAM bits* metric reports the bit count for the generated RAM blocks.

	gates	gates percentage	cells
Cluster 2 (total)	28134	100.0	8418
AIB Cache RAM	4237	15.1	
AIB Cache Control	376	1.3	129
Execute Directive Queue	908	3.2	351
Execute Directive Sequencers	1221	4.3	526
Writeback-op Queue, Bypass/Stall	1231	4.4	410
Writeback Control	582	2.1	227
Register File	10004	35.6	3013
Datapath	6694	23.8	2605
Datapath Control	2075	7.4	828
Transport Control	472	1.7	188

Table 6.3: Cluster area breakdown. The data is for cluster 2, a basic cluster with no special operations. Many of the blocks are demarcated in Figure 6-9. Columns are as in Table 6.2.

vector-thread unit uses the equivalent of around 76.6 KB of SRAM. The actual area proportions on the chip are somewhat larger than this since the standard cell regions have lower area utilization than the SRAM blocks.

Support for the vector-thread commands does not add undue complexity to the control processor. The largest area overhead is the logic to compute `vlmax` which takes up around 5% of the CP area. Even including the VT overhead, the baseline architectural components of the control processor—the 32-entry register file; the adder, logic unit, and shifter datapath components; and the iterative multiply/divide unit—occupy around half of its total area.

The vector-thread unit area is dominated by the lanes. Together, the VTU command queues, the vector load unit, the vector store unit, the vector refill unit, and the AIB Fill Unit use less than 5% of the VTU area. Within a lane, the compute clusters occupy 79% of the area, and the command management unit, the lane vector memory unit, and the store-data cluster occupy the remainder. The area of a basic cluster (c2) is dominated by the register file (36%), the datapath (24%), and the AIB cache (15%). The execute directive queue uses 3%, and the synthesized control logic makes up the remaining 22%.

It is also useful to consider a functional breakdown of the VTU area. The register files in the VTU take up 26% of its area, and the arithmetic units, including the cluster adders, logic units, shifters, and multiply/divide units, take up 9%. Together, these baseline compute and storage resources comprise 35% of the VTU area. In a vector-thread processor with floating-point units or other special compute operations, this ratio would be even higher. The vector memory access units—the LVMUs, and the VLU, VSU, VRU, and their associated command queues—occupy 6% of the total, and the memory access resources in cluster 0 and the store-data cluster add another 3%. The AIB caching resources, including the AIB cache RAMs, tags, control, and the AIB fill unit, comprise 19% of the total. The vector and thread command control overhead—the VTU command queue, the execute directive queues and sequencers, the pending thread fetch queue, and other logic in the CMU—together take up 11% of the VTU area. The writeback decoupling resources in the VTU, including the datapath latches and the bypass/stall logic for the writeback-op queue, take up 8% of the area, and the transport decoupling resources take up 4% of the area. The cross-VP queues in the lanes and the cross-VP start/stop queues together occupy 3%. The remaining 11% of the VTU area includes the cluster chain registers, predicate registers, and other pipeline latches, muxes, and datapath control logic.

6.3.2 Frequency

Figure 6-11 shows a shmoo plot of frequency versus supply voltage for the Scale chip operating in the on-chip RAM testing mode. At the nominal supply voltage of 1.8 V, the chip operates at a maximum frequency of 260 MHz. The voltage can be scaled down to 1.15 V at which point the chip runs at 145 MHz. The frequency continues to improve well beyond 1.8 V, and it reaches 350 MHz at 2.9 V (not shown in the shmoo plot). However, we have not evaluated the chip lifetime when operating above the nominal supply voltage. Of 13 working chips (out of 15), the frequency variation was generally less than 2%. We were surprised to find that using an undivided external clock input (which may have an uneven duty cycle) only reduced the maximum frequency by around 3% compared to using the VCO. Figure 6-12 shows the maximum frequency when caching is enabled (as explained in Section 6.1.7), alongside the on-chip RAM mode results. With caching, Scale's maximum frequency is 180 MHz at 1.8 V.

In the on-chip RAM operating mode, Scale's cycle time is around 43 FO4 (fan-out-of-four gate delays) based on a 90 ps FO4 metric. This is relatively competitive for an ASIC design flow, as clock periods for ASIC processors are typically 40–65 FO4 [CK02]. Our initial studies evaluated

```
            |       |       |       |       |       |       |
350 --- . . . . . . . . . . . . . . . . . . . . . . . . . . . ---
345     . . . . . . . . . . . . . . . . . . . . . . . . . . .
340     . . . . . . . . . . . . . . . . . . . . . . . . . . .
335     . . . . . . . . . . . . . . . . . . . . . . . . . . .
330     . . . . . . . . . . . . . . . . . . . . . . . . . . .
325     . . . . . . . . . . . . . . . . . . . . . . . . . . .
320     . . . . . . . . . . . . . . . . . . . . . . . . . . *
315     . . . . . . . . . . . . . . . . . . . . . . . . * * *
310     . . . . . . . . . . . . . . . . . . . . . . . * * * *
305     . . . . . . . . . . . . . . . . . . . . . . * * * * *
300 --- . . . . . . . . . . . . . . . . . . . . . * * * * * * ---
295     . . . . . . . . . . . . . . . . . . . . * * * * * * *
290     . . . . . . . . . . . . . . . . . . . * * * * * * * *
285     . . . . . . . . . . . . . . . . . . * * * * * * * * *
280     . . . . . . . . . . . . . . . . . * * * * * * * * * *
275     . . . . . . . . . . . . . . . . * * * * * * * * * * *
270     . . . . . . . . . . . . . . . . * * * * * * * * * * *
265     . . . . . . . . . . . . . . . * * * * * * * * * * * *
260     . . . . . . . . . . . . . . * * * * * * * * * * * * *
255     . . . . . . . . . . . . . * * * * * * * * * * * * * *
250 --- . . . . . . . . . . . . . * * * * * * * * * * * * * * ---
245     . . . . . . . . . . . . * * * * * * * * * * * * * * *
240     . . . . . . . . . . . * * * * * * * * * * * * * * * *
235     . . . . . . . . . . . * * * * * * * * * * * * * * * *
230     . . . . . . . . . . * * * * * * * * * * * * * * * * *
225     . . . . . . . . . * * * * * * * * * * * * * * * * * *
220     . . . . . . . . . * * * * * * * * * * * * * * * * * *
215     . . . . . . . . * * * * * * * * * * * * * * * * * * *
210     . . . . . . . . * * * * * * * * * * * * * * * * * * *
205     . . . . . . . * * * * * * * * * * * * * * * * * * * *
200 --- . . . . . . * * * * * * * * * * * * * * * * * * * * * ---
195     . . . . . . * * * * * * * * * * * * * * * * * * * * *
190     . . . . . * * * * * * * * * * * * * * * * * * * * * *
185     . . . . . * * * * * * * * * * * * * * * * * * * * * *
180     . . . . . * * * * * * * * * * * * * * * * * * * * * *
175     . . . . * * * * * * * * * * * * * * * * * * * * * * *
170     . . . . * * * * * * * * * * * * * * * * * * * * * * *
165     . . . * * * * * * * * * * * * * * * * * * * * * * * *
160     . . . * * * * * * * * * * * * * * * * * * * * * * * *
155     . . * * * * * * * * * * * * * * * * * * * * * * * * *
150 --- . . * * * * * * * * * * * * * * * * * * * * * * * * * ---
145     . * * * * * * * * * * * * * * * * * * * * * * * * * *
140     . * * * * * * * * * * * * * * * * * * * * * * * * * *
135     . * * * * * * * * * * * * * * * * * * * * * * * * * *
130     . * * * * * * * * * * * * * * * * * * * * * * * * * *
125     . * * * * * * * * * * * * * * * * * * * * * * * * * *
120     . * * * * * * * * * * * * * * * * * * * * * * * * * *
115     . * * * * * * * * * * * * * * * * * * * * * * * * * *
110     . * * * * * * * * * * * * * * * * * * * * * * * * * *
105     . * * * * * * * * * * * * * * * * * * * * * * * * * *
100 --- . * * * * * * * * * * * * * * * * * * * * * * * * * * ---
            |       |       |       |       |       |       |
        1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2
        1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 0 0 1 1 2 2 3 3 4
        0 5 0 5 0 5 0 5 0 5 0 5 0 5 0 5 0 5 0 5 0 5 0 5 0 5 0
        0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figure 6-11: Shmoo plot for Scale chip operating in the on-chip RAM testing mode. The horizontal axis plots supply voltage in mV, and the vertical axis plots frequency in MHz. A "`*`" indicates that the chip functioned correctly at that operating point, and a "`.`" indicates that the chip failed.

Scale running at 400 MHz [KBH$^+$04a], but this estimate assumed a more custom design flow. We did previously build a test-chip which ran at 450 MHz (25 FO4) using the same design flow and process technology [BKA07]. However, the test-chip was a simple RISC core that was much easier to optimize than the entire Scale design.

The timing optimization passes in the synthesis and place-and-route tools work to balance all paths around the same maximum latency, i.e. paths are not made any faster than they need to be. As a result, there are generally a very large number of "critical" timing paths. Furthermore, different paths appear critical at different stages of the timing optimization, and they affect the final timing even if they become less critical at the end. Throughout the Scale implementation effort, we continuously improved the design timing by restructuring logic to eliminate critical paths. Some of the critical paths in the design include (in no particular order):

- The single-cycle 16-bit multiplier in cluster 3.
- The `vlmax` calculation in the control processor (for a `vcfgvl` command).
- Receiving a transport send signal from another cluster, then stalling stage D of the cluster pipeline based on a register dependency, and then propagating this stall back to the execute directive sequencers (stage I).
- Performing the VRI to PRI register index translation in stage D of the cluster pipeline, then stalling based on a register dependency, and then propagating this stall back to the execute directive sequencers.

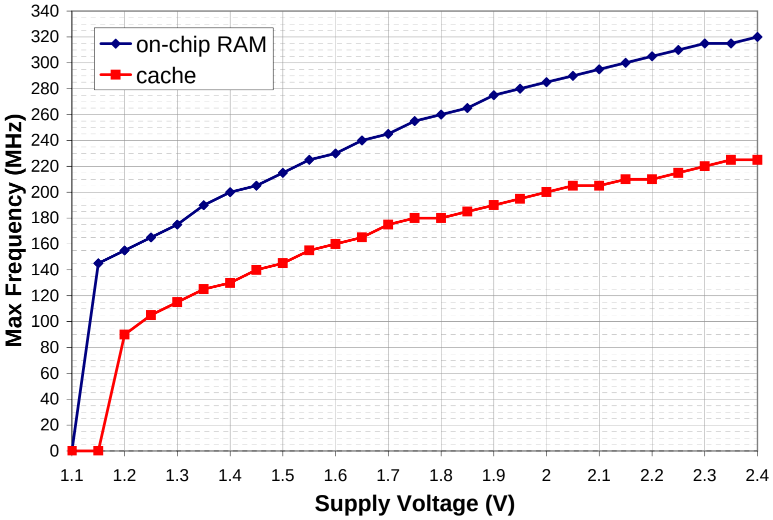


Figure 6-12: Maximum Scale chip frequency versus supply voltage for on-chip RAM mode and cache mode.

- Performing the VRI to PRI translation, then reading the register file in stage D of the cluster pipeline.

- Performing load/store address disambiguation in stage X of cluster 0, then propagating a memory request through the LVMU, and then going through cache arbitration.

- Broadcasting the read crossbar control signals, then sending load data through the crossbar, through the byte selection and sign extension unit, and to the load data queue in cluster 0.

- Determining if the command management unit can issue an execute directive (i.e. if all the clusters in the lane are ready to receive it), and then issuing a register ED from the CMU straight to stage D of a cluster pipeline.

- Reading data from the vector load data queue in the lane vector memory unit, then sending it across the length of the lane to the receive queue (latch) in cluster 3.

- Performing the AIB cache tag check (a CAM search) in a command management unit, then determining if the next VTU command can issue (if all the CMUs are ready to receive it), and then issuing the VTU command to the CMUs.

When caching is enabled, some of the additional critical paths include:

- A tag check (a CAM search in a subbank), then encoding the match signals as a RAM index, or OR-ing the match signals together to determine a store miss and select the pending store data buffer RAM index, then sending the index across the chip to the data bank.

- Reading a replay queue entry from the MSHR, then arbitrating across the banks to choose one replay, then choosing between a replay and a new access for a requester, then going through cache arbitration.

```
    la t9, input_end
forever:
    la t8, input_start
    la t7, output
loop:
    lw   t0, 0(t8)
    addu t8, 4
    bgez t0, pos
    subu t0, $0, t0
pos:
    sw   t0, 0(t7)
    addu t7, 4
    bne  t8, t9, loop
    b    forever
```

Figure 6-13: Test program for measuring control processor power consumption. The inner loop calculates the absolute values of integers in an input array and stores the results to an output array. The input array had 10 integers during testing, and the program repeatedly runs this inner loop forever.

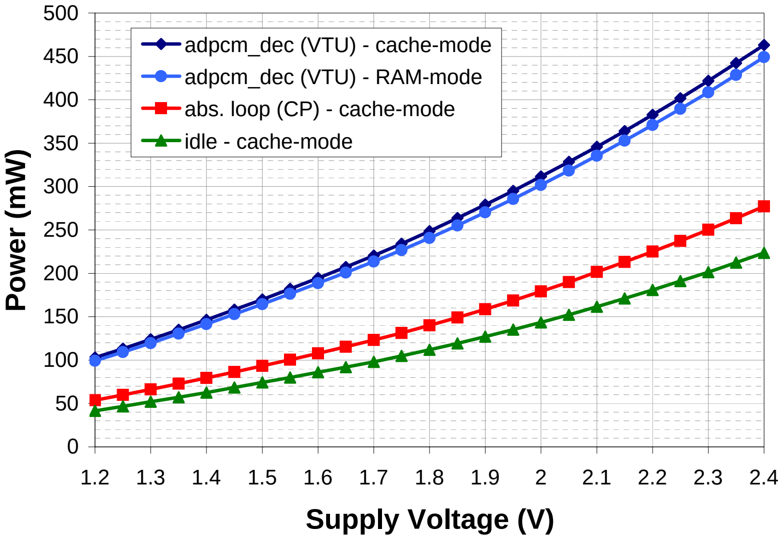


Figure 6-14: Chip power consumption versus supply voltage at 100 MHz. Results are shown for idle (clock only), the control processor absolute-value test program, and the VTU `adpcm_dec` test program. The cache is enabled for the idle and CP results. For the VTU program, results are shown for both the on-chip RAM operating mode and the cache operating mode.

6.3.3 Power

The power consumption data in this section was recorded by measuring steady-state average current draw during operation. The power measurements are for the core of the chip and do not include the input and output pads. We measure the idle power draw when the clock is running but Scale is not fetching or executing any instructions. To estimate typical control processor power consumption, we use the simple test program shown in Figure 6-13. The code fits in the control processor's L0 instruction buffer. To estimate typical power consumption for a program that uses the VTU we use the `adpcm_dec` test program described in Section 7.3.4, but we scaled down the data size to fit in the cache. This test program executes around 6.5 VTU compute-ops per cycle. Section 7.6 reports power measurements for other benchmarks.

Figure 6-14 shows how the power consumption scales with supply voltage with the chip running at 100 MHz. At 1.8 V, the idle power is 112 mW, the power with the control processor running is 140 mW, and the power with the VTU also active is 249 mW. The figure also shows the power

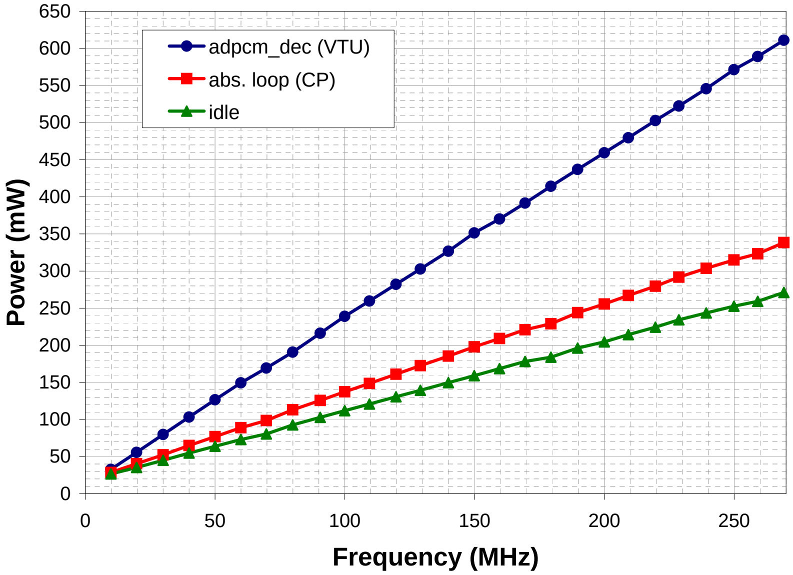


Figure 6-15: Chip power consumption versus frequency with a 1.8 V supply voltage. All data is for the on-chip RAM operating mode.

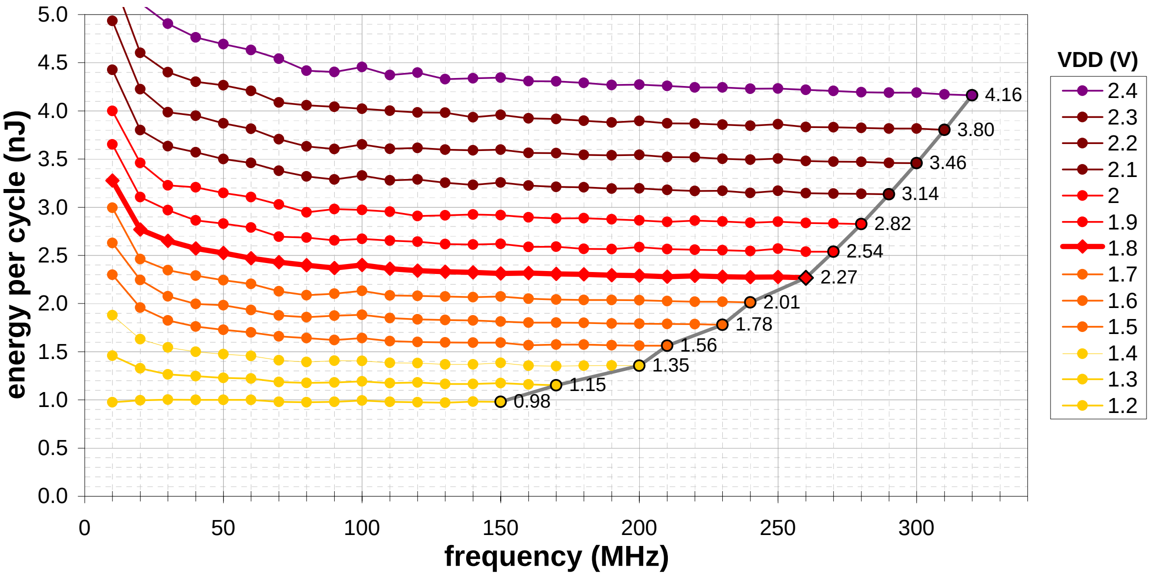


Figure 6-16: Chip energy versus frequency for various supply voltages. All data is for running the `adpcm_dec` test program in the on-chip RAM operating mode. For each supply voltage, the maximum frequency point is marked and labeled with the energy value.

for Scale operating in the on-chip RAM testing mode. At 1.8 V the difference is only 3%, and the chip uses 241 mW. The power scaling is roughly linear, but the slope increases slightly above the nominal supply voltage.

Figure 6-15 shows how the power consumption scales with frequency for the same test programs. At 260 MHz, the chip consumes 259 mW while idle, 323 mW while running the control processor program, and 589 mW while running the VTU program. The power scales linearly with frequency.

As another way to evaluate the chip power consumption, Figure 6-16 shows how energy scales with frequency for a range of supply voltages. To calculate the average energy per cycle, we multiply the average power consumption by the clock period. This is a measurement of the energy to perform a fixed amount of work (one clock cycle), so it does not tend to change very much with frequency as long as the supply voltage remains fixed. However, energy consumption can be reduced at a given frequency by lowering the supply voltage as much as possible. At low clock frequencies the chip can run at a low supply voltage, but as the speed increases the supply voltage must also increase. The outer edge of the data points in Figure 6-16 demonstrate that the minimum energy per clock cycle increases as the clock frequency increases. Another way to interpret this is that the energy used to execute a given program will increase as the chip executes the program faster. From the baseline operating point of 1.8 V and 260 MHz, the energy can be reduced by 57% (from 2.3 nJ to 1.0 nJ) if the frequency is scaled down by 42% to 150 MHz and the voltage is reduced to 1.2 V. The frequency can also be increased by 23% to 320 MHz, but the supply voltage must be raised to 2.4 V which increases the energy consumption by 83% (to 4.2 nJ).

6.4 Implementation Summary

Our tool and verification flow helped us to build a performance, power, and area efficient chip with limited resources. A highly automated methodology allowed us to easily accommodate an evolving design. Indeed, we only added the non-blocking cache miss logic to the chip during the final two months before tapeout. Our iterative approach allowed us to continuously optimize the design and to add and verify new features up until the last few days before tapeout. Even though we used a standard-cell based ASIC-style design flow, procedural datapath pre-placement allowed us to optimize timing and area. We reduced power consumption by using extensive clock-gating in both the pre-placed datapaths and the synthesized control logic.

Chapter 7

Scale VT Evaluation

In this chapter we evaluate Scale's ability to flexibly expose and exploit parallelism and locality. We have mapped a selection of 32 benchmark kernels to illustrate and analyze the key features of Scale. These include examples from linear algebra, network processing, image processing, cryptography, audio processing, telecommunications, and general-purpose computing. Additionally, kernels are combined to create larger applications including a JPEG image encoder and an IEEE 802.11a wireless transmitter. Several benchmarks are described in detail in order to highlight how different types of parallelism map to the architecture, from vectorizable loops to free-running threads. We present detailed performance breakdowns and energy consumption measurements for Scale, and we compare these results to a baseline RISC processor running compiled code (i.e. the Scale control processor). We also present detailed statistics for Scale's control and data hierarchies in order to evaluate the architecture's ability to exploit locality and amortize overheads.

7.1 Programming and Simulation Methodology

Results in this chapter and the next are based on a detailed cycle-level, execution-driven simulator. The simulator includes accurate models of the VTU and the cache, but the control processor is based on a simple single-instruction-per-cycle model with zero-cycle cache access latency. Although its pipeline is not modeled, the control processor does contend for the cache and stall for misses. Since the off-chip memory system is not a focus of this work, main memory is modeled as a simple pipelined magic memory with configurable bandwidth and latency. Section 7.6 validates the performance results of the microarchitectural simulator against the Scale chip.

The Scale microarchitectural simulator is a flexible model that allows us to vary many parameters, and some of the defaults are summarized in Table 7.1. The simulator is configured to model the Scale prototype processor, with the exception that the cache uses a no-write-allocate policy that is not implemented in the Scale chip. An intra-lane transport from one cluster to another has a latency of one cycle (i.e. there will be a one cycle bubble between the producing instruction and the dependent instruction). Cross-VP transports are able to have zero cycle latency because the clusters are physically closer together and there is less fan-in for the receive operation. Cache accesses have a two cycle latency (two bubble cycles between load and use). The main memory bandwidth is 8 bytes per processor clock cycle, and cache misses have a minimum load-use latency of 35 cycles. This configuration is meant to model a 64-bit DDR DRAM port running at the same data rate as Scale, for example 400 Mbps/pin with Scale running at 400 MHz. Longer-latency memory system configurations are evaluated in Chapter 8.

For the results in this thesis, all VTU code was hand-written in Scale assembly language (as in

Vector-Thread Unit	
Number of lanes	4
Clusters per lane	4
Registers per cluster	32
AIB cache slots per cluster	32
Intra-cluster bypass latency	0 cycles
Intra-lane transport latency	1 cycle
Cross-VP transport latency	0 cycles

L1 Unified Cache	
Size	32 KB
Associativity	32 (CAM tags)
Line size	32 B
Banks	4
Maximum bank access width	16 B
Store miss policy	no-write-allocate/write-back
Load-use latency	2 cycles

Main Memory System	
Bandwidth	8 B per processor cycle
Minimum load-use latency	35 processor cycles

Table 7.1: Default parameters for Scale simulations.

Figure 4-6) and linked with C code compiled for the control processor using `gcc`. A vector-thread compiler is in development, and it is described briefly in Section 9.2.

7.2 Benchmarks and Results

Table 7.2 shows benchmark results from the Embedded Microprocessor Benchmark Consortium (EEMBC); Table 7.3 shows benchmark results from MiBench [GRE^{+}01], Mediabench [LPMS97], and SpecInt; and, Table 7.4 shows benchmark results from the Extreme Benchmark Suite (XBS) [Ger05]. All of the benchmark results show Scale's absolute performance in terms of the number of cycles required to process the given input. The results also show the per-cycle rate of VTU compute operations, multiplies, load elements, and store elements. These statistics help to evaluate the efficiency of the Scale mappings. The memory bandwidth statistic demonstrates cases where Scale is able to simultaneously sustain high compute and memory throughputs, or cases where performance is limited by the memory bandwidth. The XBS benchmarks, in particular, have results for both small data sets that fit in cache and larger data sets which stream from memory.

In addition to providing the absolute Scale performance and the number of operations executed per cycle, the results also indicate speedups compared to a scalar RISC machine. These are obtained by running compiled code on the Scale control processor. These speedups, on balance, provide a general idea of Scale's performance compared to a RISC architecture, but one should be careful in making the comparison. Although the Scale results match the typical way in which embedded processors are used, they illustrate the large speedups possible when algorithms are extensively tuned for a highly parallel processor versus compiled from standard reference code. The control processor model itself is a somewhat idealistic comparison point since it executes one instruction per cycle (subject to cache miss stalls), and it does not suffer from branch mispredictions or load-use latency for cache hits. Its MIPS-like instruction set provides compute and logic operations that are similar to those for the Scale clusters. However, it does not have an analogue to Scale's 16-bit multiply instruction (`mulh`). The compiled code either uses a `mult`/`mflo` sequence (which each execute in a single cycle in the simulator), or it executes multiplies by constants as a series of shifts

			Iterations per				Per-cycle statistics						Memory
			million cycles		Kernel	Avg			Load	Store	Mem	Loop	access
Benchmark	Description	Input	RISC	Scale	speedup	VL	Ops	Muls	Elms	Elms	Bytes	types	types
rgbcmy	RGB to CMYK color conversion	225 KB image	0.35	8.25	23.7	40.0	10.8	–	1.9	0.6	4.7	DP	VM,SVM
rgbyiq	RGB to YIQ color conversion	225 KB image	0.14	5.78	39.9	28.0	9.3	4.0	1.3	1.3	4.2	DP	SVM
hpg	High pass gray-scale filter	75 KB image	0.26	12.80	50.1	63.6	10.7	3.9	2.9	1.0	3.3	DP	VM,VP
text	Control language parsing	323 rule strings (14.0 KB)	0.82	1.21	1.5	6.7	0.3	–	0.1	0.0	–	DE	VM
dither	Floyd-Steinberg gray-scale dithering	64 KB image	0.39	2.70	7.0	30.0	5.1	0.7	1.1	0.3	0.3	DP,DC	VM,SVM,VP
rotate	Binary image 90 degree rotation	12.5 KB image	2.01	44.08	21.9	12.3	10.8	–	0.6	0.6	–	DP	VM,SVM
lookup	IP route lookup using Patricia Trie	2000 searches	4.65	33.86	7.3	40.0	6.1	–	1.3	0.0	–	DI	VM,VP
ospf	Dijkstra shortest path first	400 nodes × 4 arcs	18.81	17.78	0.9	28.0	1.2	–	0.2	0.1	–	FT	VP
pktflow	IP packet processing	512 KB (11.7 KB headers)	16.87	332.63	19.7	46.3	8.1	–	1.6	0.1	–	DC	VM,VP
		1 MB (22.5 KB headers)	6.02	89.47	14.9	51.2	4.2	–	0.8	0.1	4.7		
		2 MB (44.1 KB headers)	3.07	46.49	15.1	52.2	4.3	–	0.9	0.1	4.8		
pntrch	Searching linked list	5 searches, ≈ 350 nodes each	24.85	96.28	3.9	13.0	2.3	–	0.3	0.0	–	FT	VP
fir-hl	Finite impulse response filters	3 high + 3 low, 35-tap	135.12	15600.62	115.5	60.0	6.8	3.3	1.7	0.1	0.5	DP	VM,SVM
fbital	Bit allocation for DSL modems	step (20 carriers)	32.73	755.30	23.1	20.0	2.3	–	0.4	0.0	–	DC,XI	VM,VP
		pent (100 carriers)	3.38	163.01	48.2	100.0	3.4	–	0.5	0.0	–		
		typ (256 carriers)	2.23	60.45	27.1	85.3	3.6	–	0.5	0.3	–		
fft	256-pt fixed-point complex FFT	*all* (1 KB)	17.23	302.32	17.5	25.6	3.8	1.2	1.7	1.4	–	DP	VM,SVM
viterb	Soft decision Viterbi decoder	*all* (688 B)	3.94	40.41	10.3	16.0	4.9	0.4	0.5	0.5	–	DP	VM,SVM
autocor	Fixed-point autocorrelation	data1 (16 samples, 8 lags)	707.01	9575.15	13.5	8.0	3.9	1.2	1.4	0.1	–	DP	VM
		data2 (1024 samples, 16 lags)	4.72	161.26	34.1	16.0	7.9	2.6	2.8	0.0	–		
		data3 (500 samples, 32 lags)	4.95	197.97	40.0	32.0	9.5	3.2	3.3	0.0	–		
conven	Convolutional encoder	data1 (512 B, 5 xors/bit)	7.62	7169.28	940.9	32.0	11.0	–	0.9	0.2	–	DP	VM,VP
		data2 (512 B, 5 xors/bit)	8.87	8815.78	994.4	32.0	11.3	–	1.1	0.3	–		
		data3 (512 B, 3 xors/bit)	11.21	10477.57	935.0	32.0	10.1	–	1.3	0.3	–		

Table 7.2: Performance and mapping characterization for benchmarks from the Embedded Microprocessor Benchmark Consortium (EEMBC). The *RISC* column shows performance for compiled out-of-the-box code (OTB) running on the Scale control processor. The *Scale* column shows results for assembly-optimized code (OPT) which makes use of the VTU. The kernels were run repeatedly in a loop, and the performance metric shows the average throughput as iterations per million cycles. For EEMBC benchmarks with multiple data sets, *all* indicates that results were similar across the inputs. The *Avg VL* column displays the average vector length used by the benchmark. The *Ops* column shows VTU compute operations per cycle, the *Muls* column shows VTU multiplies per cycle (a subset of the total compute-ops), the *Load-Elm* and *Store-Elm* columns show VTU load and store elements per cycle, and the *Mem-Bytes* column shows the DRAM bandwidth in bytes per cycle. The *Loop Type* column indicates the categories of loops which were parallelized when mapping the benchmarks to Scale: [DP] data-parallel loop with no control flow, [DC] data-parallel loop with conditional thread-fetches, [XI] loop with cross-iteration dependencies, [DI] data-parallel loop with inner-loop, [DE] loop with data-dependent exit condition, and [FT] free-running threads. The *Mem Access* columns indicate the types of memory accesses performed: [VM] unit-stride and strided vector memory accesses, [SVM] segment vector memory accesses, and [VP] individual VP loads and stores.

				Total cycles				Per-cycle statistics					Memory
				(millions)		Kernel	Avg		Load	Store	Mem	Loop	access
Benchmark	**Suite**	**Description**	**Input**	**RISC**	**Scale**	**speedup**	**VL**	**Ops**	**Elms**	**Elms**	**Bytes**	**types**	**types**
rijndael	MiBench	Advanced Encryption Standard	3.1 MB	410.6	168.4	3.3	4.0	3.1	1.0	0.1	–	DP	VM,VP
sha	MiBench	Secure hash algorithm	3.1 MB	141.6	60.8	2.3	5.5	1.8	0.3	0.1	–	DP,XI	VM,VP
qsort	MiBench	Quick sort of strings	10 k strings (52.2 KB)	32.4	20.6	3.0	12.0	1.9	0.4	0.3	2.5	FT	VP
adpcm_enc	Mediabench	ADPCM speech compression	144 K samples (288 KB)	7.0	3.9	1.8	90.9	2.0	0.1	0.0	–	XI	VM,VP
adpcm_dec	Mediabench	ADPCM speech decompression	144 K samples (72 KB)	5.9	0.9	7.6	99.9	6.5	0.6	0.2	–	XI	VM,VP
li	SpecInt95	Lisp interpreter	test (queens 8)	1333.4	1046.3	5.3	62.7	2.7	0.3	0.2	2.9	DP,DC,XI,DE	VM,VP

Table 7.3: Performance and mapping characterization for miscellaneous benchmarks. Total cycle numbers are for the entire application, while the remaining statistics are for the kernel only (the kernel excludes benchmark overhead code and for `li` the kernel consists of the garbage collector only). Other columns are as described in Table 7.2.

			Input elements				Per-cycle statistics						Memory
			per 1000 cycles		Kernel	Avg			Load	Store	Mem	Loop	access
Benchmark	**Description**	**Input**	**RISC**	**Scale**	**speedup**	**VL**	**Ops**	**Muls**	**Elms**	**Elms**	**Bytes**	**types**	**types**
rgbycc	RGB to YCC color conversion	1 k pixels (2.9 KB)	11.17	434.36	38.9	31.2	9.1	3.9	1.3	1.3	–	DP	VM,SVM
		100 k pixels (293.0 KB)	10.77	443.60	41.2	32.0	9.3	4.0	1.3	1.3	4.6		
fdct	Forward DCT of 8×8 blocks	32 blocks (4 KB)	0.39	11.47	29.2	9.5	10.7	2.2	1.5	1.5	–	DP	VM,SVM
		1000 blocks (125 KB)	0.35	9.52	27.0	9.6	8.9	1.8	1.2	1.2	5.0		
idct	Inverse DCT of 8×8 blocks	32 blocks (4 KB)	0.33	11.00	32.9	9.5	11.3	2.1	1.4	1.4	–	DP	VM,SVM
		1000 blocks (125 KB)	0.30	9.54	31.3	9.6	9.8	1.8	1.2	1.2	5.2		
quantize	JPEG quantization of 8×8 blocks	32 blocks (4 KB)	0.59	1.76	3.0	113.8	1.4	–	0.3	0.1	–	DP	VM
		1000 blocks (125 KB)	0.53	1.76	3.3	119.9	1.4	–	0.3	0.1	0.8		
jpegenc	JPEG compression of 8×8 blocks	1 block (128 B)	0.47	1.29	2.8	3.9	1.7	–	0.2	0.0	–	XI,DE	VM,VP
scrambler	802.11a bit scrambler	4 k bytes	477.70	3087.36	6.5	11.9	3.6	–	0.8	0.8	–	DP	SVM
		100 k bytes	314.20	3300.93	10.5	12.0	3.7	–	0.8	0.8	7.7		
convolenc	802.11a convolutional encoder	4 k bytes	33.33	276.19	8.3	54.1	4.6	–	0.7	0.1	–	DP	VM,VP
		100 k bytes	32.11	276.93	8.6	55.9	4.6	–	0.7	0.1	0.9		
interleaver	802.11a bit interleaver	4.8 k bytes	22.29	540.54	24.2	33.3	13.9	–	0.1	0.5	–	DP	SVM
		96 k bytes	21.75	538.45	24.8	35.7	13.8	–	0.1	0.5	3.8		
mapper	802.11a mapper to OFDM symbols	1.2 k bytes	15.29	544.95	35.6	16.7	3.8	–	1.4	1.5	–	DP	VM,SVM,VP
		48 k bytes	15.06	419.99	27.9	20.0	2.9	–	1.1	1.1	7.7		
ifft	64-pt complex-value inverse FFT	32 blocks (8 KB)	0.06	1.45	25.1	16.0	11.0	1.1	0.4	0.4	–	DP	VM,SVM
		1000 blocks (250 KB)	0.06	1.23	21.7	16.0	9.3	0.9	0.4	0.3	4.5		

Table 7.4: Performance and mapping characterization for benchmarks from the Extreme Benchmark Suite (XBS). Each kernel has one data set which fits in cache and one which does not. The kernels were run repeatedly in a loop, and the performance metric shows the average throughput as input elements (pixels, blocks, or bytes) processed per thousand cycles. Other columns are as described in Table 7.2.

and adds.

The EEMBC rules indicate that benchmarks may either be run "out-of-the-box" (OTB) as compiled unmodified C code, or they may be optimized (OPT) using assembly coding and arbitrary hand-tuning. One drawback of this form of evaluation is that performance depends greatly on programmer effort, especially as EEMBC permits algorithmic and data-structure changes to many of the benchmark kernels, and optimizations used for the reported results are often unpublished. We made algorithmic changes to several of the EEMBC benchmarks: `rotate` blocks the algorithm to enable rotating an 8-bit block completely in registers, `pktflow` implements the packet descriptor queue using an array instead of a linked list, `fir-hl` optimizes the default algorithm to avoid copying and exploit reuse, `fbital` uses a binary search to optimize the bit allocation, `conven` uses bit packed input data to enable multiple bit-level operations to be performed in parallel, and `fft` uses a radix-2 hybrid Stockham algorithm to eliminate bit-reversal and increase vector lengths. These algorithmic changes help Scale to achieve some of the very large speedups reported in Table 7.2.

For the other benchmark suites, the algorithms used in the Scale mapping were predominately the same as the C code used for the RISC results. For the XBS benchmarks, the optimized C versions of the algorithms were used for the RISC results [Ger05].

Overall, the results show that Scale can flexibly exploit parallelism on a wide range of codes from different domains. The tables indicate the categories of loops which were parallelized when mapping the benchmarks to Scale. Many of the benchmarks have data-parallel vectorizable loops (DP), and Scale generally achieves very high performance on these; it executes more than 8 compute-ops per cycle on 11 out of 32 of the benchmarks, and the speedups compared to RISC are generally 20–40×. This performance is often achieved while simultaneously sustaining high memory bandwidths, an effect particularly seen with the XBS benchmarks. Other data-parallel loops in the benchmarks either have large conditional blocks (DC), e.g. `dither`, or inner loops (DI), e.g. `lookup`. VP thread fetches allow these codes to execute efficiently on Scale; they achieve 5–6 compute-ops per cycle and a speedup of about 7×. The benchmarks that have loops with cross-iteration dependencies (XI) tend to have more limited parallelism, but Scale achieves high throughput on `adpcm_dec` and moderate speedups on `fbital`, `adpcm_enc`, `sha`, and `jpegenc`. Finally, Scale uses a free-running VP threads mapping for `qsort` and `pntrch` to achieve 3–4× speedups, despite `qsort`'s relatively high cache miss rate and the structure of `pntrch` which artificially limits the maximum parallelism to 5 searches. The following section describe several of the benchmarks and their Scale mappings in greater detail.

Tables 7.2, 7.3, and 7.4 also indicate the types of memory accesses used by the Scale benchmark mappings: unit-stride and strided vector memory accesses (VM), segment vector memory accesses (SVM), and individual VP loads and stores (VP). Segment accesses are unique to Scale and they work particularly well when vector operations are grouped into AIBs. They expose parallelism more explicitly than the overlapping strided loads and stores that a traditional vector architecture would use for two-dimensional access patterns. Many algorithms process neighboring data elements together, and 14 out of 32 of the benchmarks make use of multi-element segment vector memory accesses. The memory access characterization in the tables also demonstrates how many non-vectorizable kernels—those with cross-iteration dependencies or internal control flow—can still take advantage of efficient vector memory accesses.

7.3 Mapping Parallelism to Scale

This section provides an overview of the Scale VT architecture's flexibility in mapping different types of code. Through several example benchmark mappings, we show how Scale executes vector-

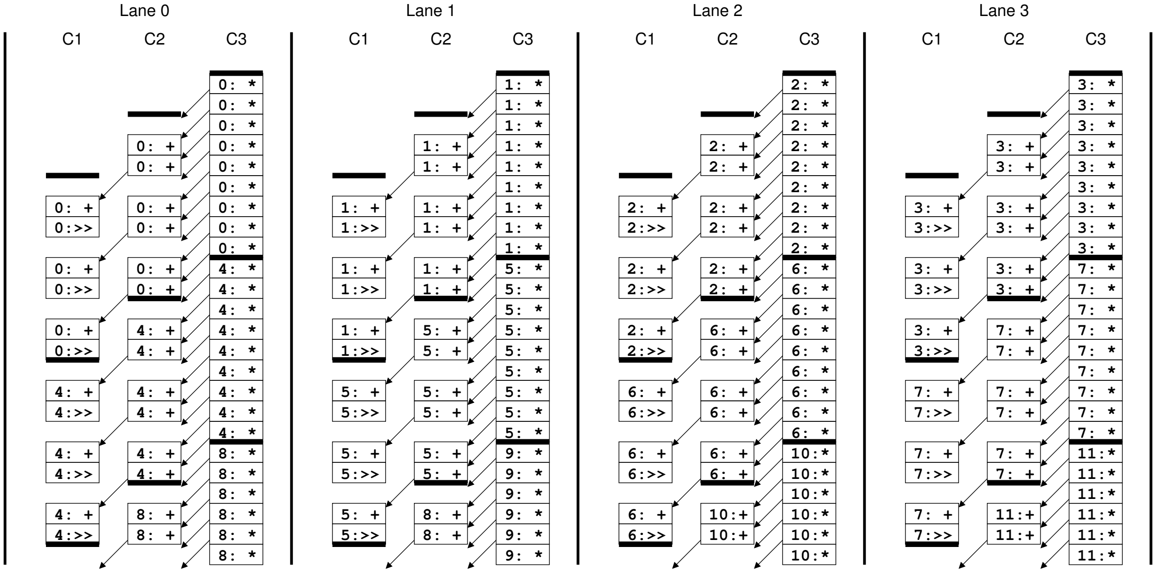


Figure 7-1: Execution of `rgbycc` kernel on Scale. VP numbers for the compute-ops are indicated before the colon. Time progresses downwards in the figure.

izable loops, loops with internal control flow, loops with cross-iteration dependencies, and reduction operations. This section also describes how free-running threads are used when structured loop-level parallelism is not available.

7.3.1 Data-Parallel Loops with No Control Flow

Data-parallel loops with no internal control flow, i.e. simple vectorizable loops, may be ported to Scale in a similar manner as other vector architectures. Vector-fetch commands encode the cross-iteration parallelism between blocks of instructions, while vector memory commands encode data locality and enable optimized memory access. The image pixel processing benchmarks (`rgbcmy`, `rgbyiq`, `hpg`, `rotate`, and `rgbycc`) are examples of streaming vectorizable code for which Scale is able to achieve high throughput.

RGB-to-YCC Example. An example vectorizable loop is the `rgbycc` kernel. The Scale AIB is similar to the example shown in Figure 4-2. The RGB bytes are read from memory with a 3-element segment load. To transform a pixel from RGB to each of the three output components (Y, Cb, and Cr), the vector-fetched AIB uses three multiplies, three adds, and a shift. The Scale code in the AIB processes one pixel—i.e., one loop iteration—in a straightforward manner with no cross-iteration instruction scheduling. The outputs are then written to memory using three unit-stride vector-stores.

The scaling constants for the color conversion are held in shared registers. The example in Figure 4-2 uses 9 shared registers, but the actual Scale implementation uses only 8 since two of the scaling constants are the same. These are written by the control processor, and they take the place of vector-scalar instructions in a traditional vector machine. If each VP had to keep its own copy of the constants, a 32-entry register file could only support 2 VPs (with 11 registers each). However, with 8 shared registers and 3 private registers, the same register file can support 8 VPs, giving a vector length of 32 across the lanes.

Within the color conversion computation for each output component, all temporary values are held in chain registers. In fact, even though the AIB contains 21 VP instructions, only the three final

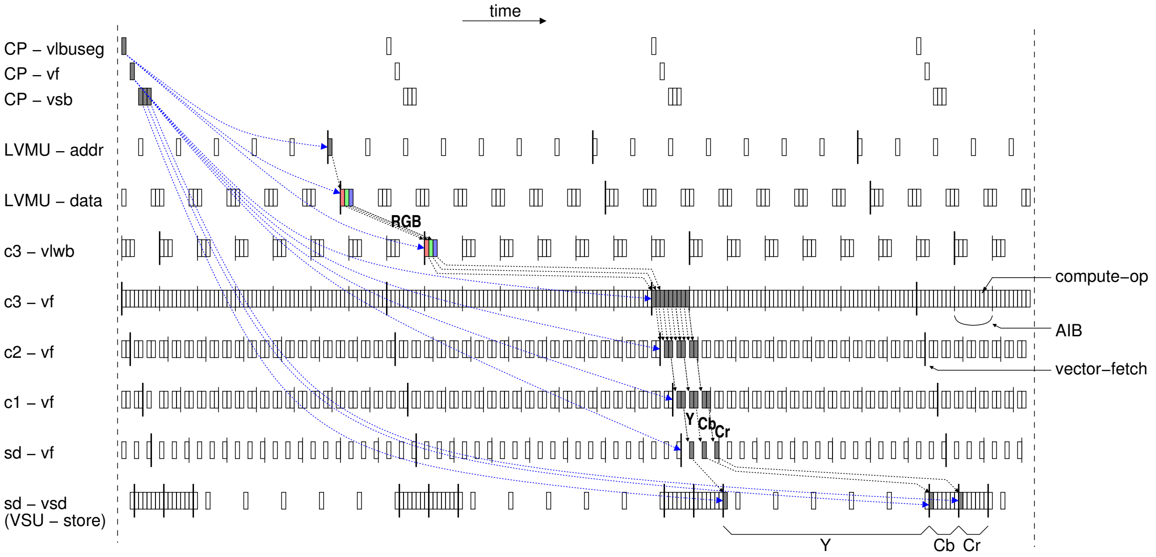
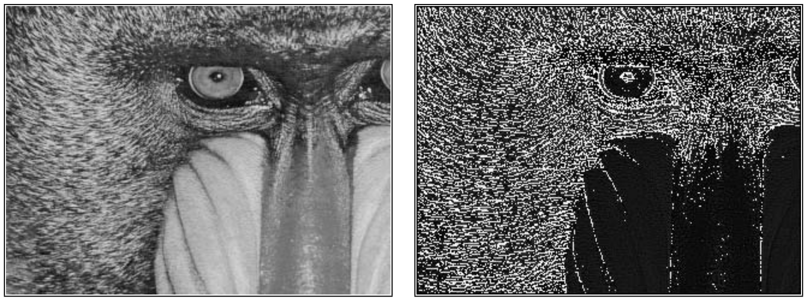


Figure 7-2: Diagram of decoupled execution for `rgbycc` kernel. One lane is shown. The operations involved in a single pixel transformation are highlighted. The control processor would generally be running even further ahead, but the decoupling has been condensed to fit on the figure.

output values (14%) are written to a register file (the private store-data registers). Of the 42 input operands for VP instructions, 21 (50%) come from private and shared registers, 18 (43%) come from chain registers, and 3 (7%) are immediates.

The per-cluster micro-op bundles for the AIB are shown in Figure 5-5, and Figure 7-1 shows a diagram of the kernel executing on Scale. The bottleneck is cluster 3, and it continually executes multiplies and sends the results to cluster 2. Cluster 2 accumulates the data and forwards the sums to cluster 1 for scaling and rounding. The clusters in Scale are decoupled—although each individual cluster executes its own micro-ops in-order, the inter-cluster execution slips according to the data dependencies. In the example, the inter-cluster dependencies are acyclic, allowing c3 to run ahead and execute instructions from the next VP on the lane before c2 and c1 have finished the current one. This scheduling is done dynamically in hardware based on simple inter-cluster dataflow between the transport and writeback micro-ops (which are explicit in Figure 5-5). The cluster decoupling hides the one-cycle inter-cluster transport latency as well as the latency of the 16-bit multiplier (although the Scale multiplier executes in a single cycle, the result can not be directly bypassed to a dependent instruction in cluster 3). In a traditional vector machine, chaining allows different functional units to operate concurrently. In Scale, the operations within an AIB execute concurrently across the clusters.

Figure 7-2 shows a broader timeline diagram of the `rgbycc` kernel executing on Scale. During the execution, the control processor runs ahead, queuing the vector-load, vector-fetch, and vector-store commands for these decoupled units. The segment vector-loads get distributed to the lane vector memory units (LVMUs) which run ahead and fetch data into their vector load data queues (Figure 5-19); each cache access fetches 3 byte elements (R, G, and B). Corresponding vector-load writeback execute directives (vlwb EDs) on cluster 3 receive the load data and write it to private registers. Since c3 executes 9 multiplies for every 3 load elements, the vector-fetch EDs limit throughput and the vlwb EDs end up chaining behind, similar to the chaining in Figure 5-15. The progress of c3 throttles all of the other decoupled units—the command and data queues for



(a) hpg input (b) hpg output

Figure 7-3: Inputs and outputs for the hpg benchmark. The images are gray-scale with 8 bits per pixel.

preceding units eventually fill up, causing them to stall, and the subsequent units must wait for data to be available.

After c3, c2, and c1 complete the color conversion for a pixel, the three output components are written to private store-data registers in sd. For each vector-fetch, three vector-stores are used to write the data to memory. The vector-store unit (VSU) orchestrates the unit-stride accesses, and it combines data across the 4 lanes for each cache write. The first vector-store (for the Y outputs) chains on the vector-fetch in the sd cluster. After it completes, the other two vector-stores (for the Cb and Cr outputs) execute in bursts. The rgbycc AIB could be partitioned into three blocks to interleave the three vector-stores with the computation, but in this example the stores are not a bottleneck.

For the rgbycc kernel, chaining allows the vector load data writebacks and the vector store data reads to essentially operate in the background while the computation proceeds with no stalls. Decoupling hides any cache bank conflicts or stall conditions in the vector memory units or the control processor. Scale is able to achieve its peak sustained throughput of 4 multiplies per cycle. The VTU executes an average of 9.33 compute-ops per cycle, while simultaneously fetching an average of 1.33 load elements per cycle and writing 1.33 store elements per cycle (an equivalent of 12 total operations per cycle). The vector memory commands reduce cache access bandwidth: each read fetches the 3 load elements from a segment, and each write combines 4 store elements from across the lanes. Thus, the VTU performs around 0.78 cache accesses per cycle, a bandwidth easily satisfied by Scale's 4-bank cache. Overall, Scale does the RGB-to-YCC conversion at a rate of 2.25 cycles per pixel, and it processes an image at a rate of 1.33 bytes per cycle.

High Pass Gray-Scale Filter Example. The hpg benchmark from the EEMBC suite computes a high pass filter of a gray-scale image encoded with one byte per pixel. Each output pixel is calculated as a weighted average of 9 pixels in the input image: its corresponding pixel and the 8 surrounding neighbor pixels. To take the average, the corresponding pixel is multiplied by a "CENTER" coefficient, and the neighboring pixels are multiplied by "HALO" coefficients. In the benchmark these coefficients are weighted to detect edges; the effect is shown in Figure 7-3.

In the Scale mapping of hpg, each virtual processor calculates a column of output pixels. The mapping takes advantage of data and computation reuse that is inherent in the algorithm. As part of the calculation, three neighboring pixels are each multiplied by the HALO parameter and summed together. This sum is used for the bottom row of one output pixel, and then it is reused as the top row of the output pixel two rows below. Figure 7-4 shows a diagram of the mapping. A straightforward mapping would perform 9 loads, 9 multiplies, 8 additions, and 1 store for each output pixel. The

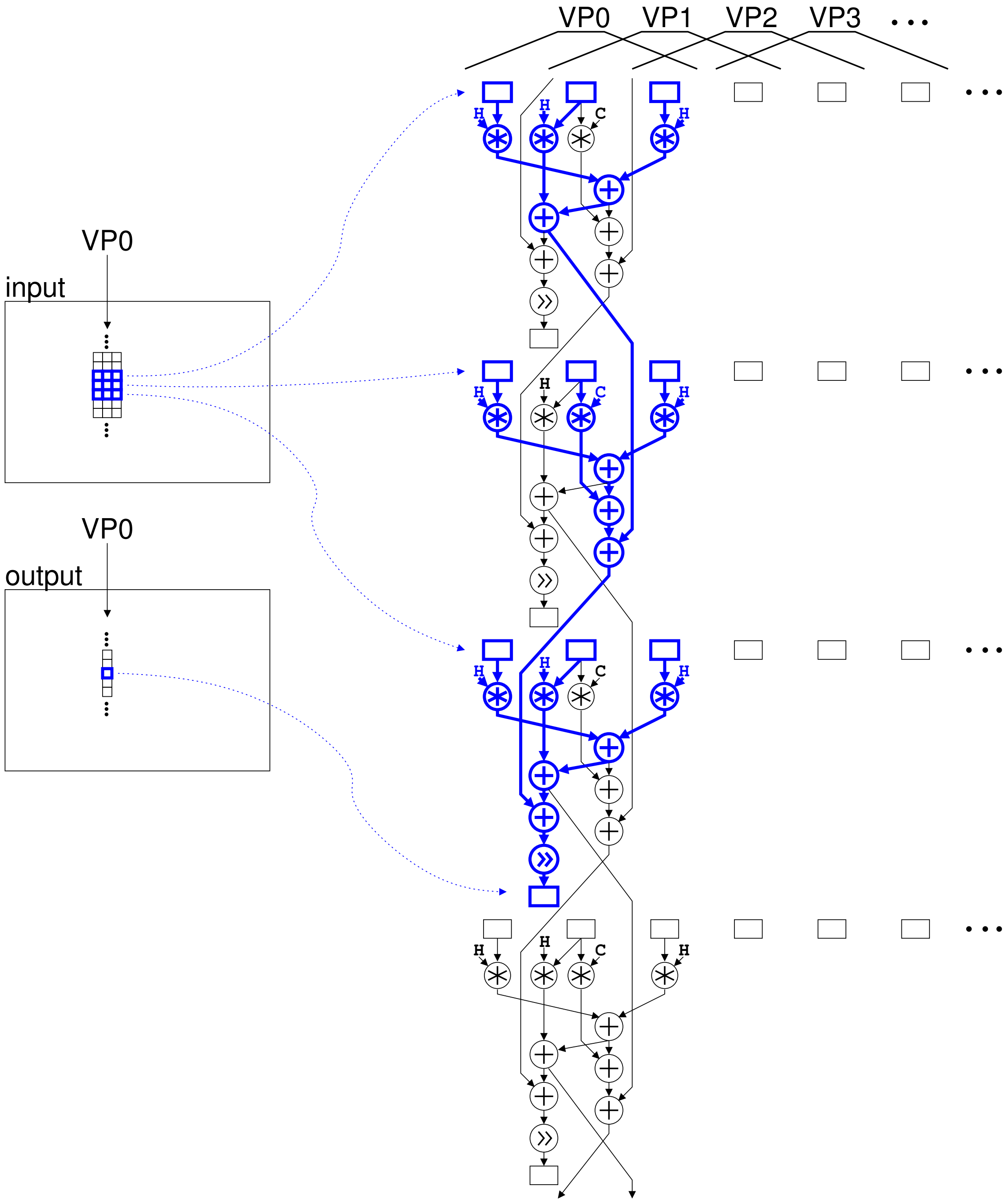


Figure 7-4: Diagram of hpg mapping. The compute operations for one virtual processor (VP0) are shown. Each step has 3 input pixels and 1 output pixel. The computation for one output pixel stretches over three steps, as highlighted in the figure.

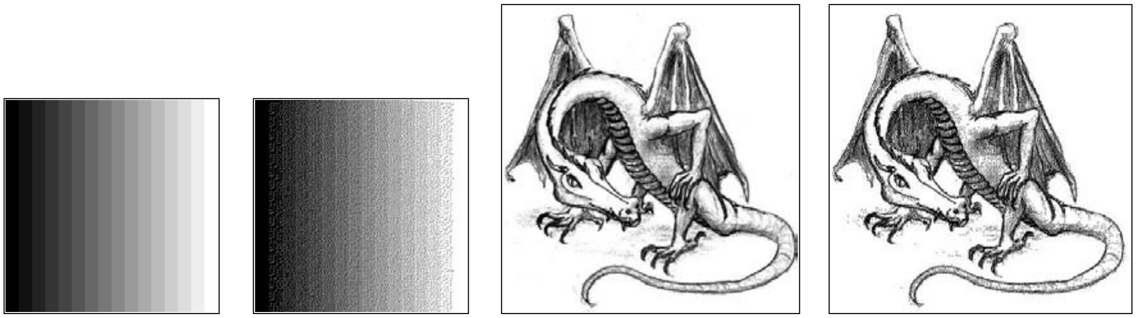
```
[ ... ]
  vcfgvl zero, zero, 2,0, 1,1, 1,0, 1,2
[ ... ]
loop:
  vlbu inrowp, c3/pr0
  vf   vtu_left
  addu t0, inrowp, 2
  vlbu t0, c3/pr0
  vf   vtu_right
  addu t0, inrowp, 1
  vlbu t0, c3/pr0
  vf   vtu_center
  vsbai outrowp, cols, c0/sd0
  addu inrowp, cols
  subu outrows, 1
  bnez outrows, loop
[ ... ]

vtu_left:
  .aib begin
  c3 mulh pr0, sr0 -> c2/pr0   # L*HALO
  c0 copy pr0      -> c1/pr0   # MiddleTop = nextMiddleTop
  .aib end

vtu_right:
  .aib begin
  c3 mulh pr0, sr0 -> c2/cr1   # R*HALO
  c2 addu pr0, cr1 -> pr0      # (L+R)*HALO
  .aib end

vtu_center:
  .aib begin
  c3 mulh pr0, sr1 -> c2/cr0   # M*CENTER
  c2 addu cr0, pr0 -> c0/cr0   # nextMiddle = (L+R)*HALO + M*CENTER
  c0 addu cr0, pr1 -> pr0      # nextMiddleTop = nextMiddle + prevBottom
  c3 mulh pr0, sr0 -> c2/cr0   # M*HALO
  c2 addu cr0, pr0 -> c1/cr0, c0/pr1 # Bottom = (L+M+R)*HALO
  c1 addu cr0, pr0 -> cr0      # All = Bottom + MiddleTop
  c1 sra  cr0, 8   -> c0/sd0
  .aib end
```

Figure 7-5: Scale code for hpg. The code shown computes a column of the output image, with a width equal to the vector length. Not shown are the outer loop which computes the other columns in the image, or the code which handles the edges of the image.



(a) gradation input (b) gradation output (c) dragon input (d) dragon output

Figure 7-6: Inputs and outputs for the `dither` benchmark. The inputs are gray-scale images with 8 bits per pixel, and the outputs are binary images with 1 bit per pixel.

		White	Compute-ops per pixel		Pixels per 1000 cycles	
Input	Size	pixels	Scale-pred	Scale-branch	Scale-pred	Scale-branch
gradation	64 KB	6.3%	27	28.3	181	178
dragon	139 KB	63.8%	27	12.8	211	289

Table 7.5: Performance comparison of branching and predication for the `dither` benchmark. The two inputs differ in their ratio of white pixels. The performance metric is the number of pixels processed per 1000 clock cycles.

Scale VPs exploit the reuse to perform only 3 loads, 4 multiplies, 5 additions, and 1 store for each output.

Figure 7-5 shows an overview of the Scale code. Each step uses three overlapping unit-stride vector-loads to fetch the input pixels and a unit-stride vector-store to write the output pixels. The computation within a step is split into three AIBs to enable chaining on the vector-loads. The two values which cross between stages of the computation are held in private VP registers (c0/pr0 and c0/pr1). The CENTER and HALO constants are held in shared registers, and chain register are used to hold the temporaries between operations. The vector length is limited by the two private registers on cluster 0, and the 4 lanes are able to support 64 VPs.

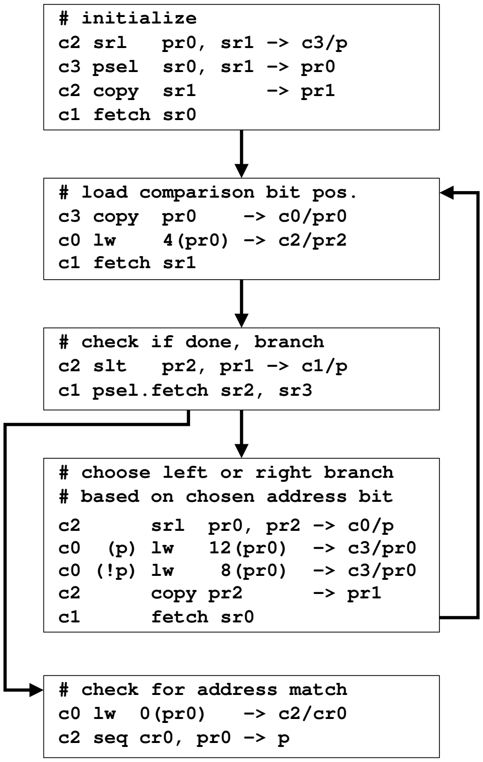
For this kernel, the cache acts as an important memory bandwidth amplifier. The overlapping vector-loads fetch each input byte three times, but the cache exploits the temporal locality so that each byte is only fetched once from external memory.

The `hpg` kernel requires higher load element bandwidth than RGB-to-YCC, but the performance is still limited by the multipliers on Scale. The peak of 4 multiplies per cycle results in an average of 10 total compute-ops per cycle, plus 3 load elements and 1 store element per cycle. Including the outer-loop vectorization overheads and the handling of the edges of the image, Scale is able to achieve around 96% of the theoretical peak. Overall, the filter runs at a rate of 0.96 pixels per cycle.

7.3.2 Data-Parallel Loops with Conditionals

Traditional vector machines handle conditional code with predication (masking), but the VT architecture adds the ability to conditionally branch. Predication can be less overhead for small conditionals, but branching results in less work when conditional blocks are large.

The EEMBC `dither` benchmark includes an example of a large conditional block in a data parallel loop. As shown in Figure 7-6, the benchmark converts gray-scale images to black and



```
# initialize
c2 srl   pr0, sr1 -> c3/p
c3 psel  sr0, sr1 -> pr0
c2 copy  sr1      -> pr1
c1 fetch sr0
```

```
# load comparison bit pos.
c3 copy  pr0    -> c0/pr0
c0 lw    4(pr0) -> c2/pr2
c1 fetch sr1
```

```
# check if done, branch
c2 slt   pr2, pr1 -> c1/p
c1 psel.fetch sr2, sr3
```

```
# choose left or right branch
# based on chosen address bit
c2       srl  pr0, pr2 -> c0/p
c0  (p)  lw   12(pr0)  -> c3/pr0
c0 (!p)  lw    8(pr0)  -> c3/pr0
c2       copy pr2      -> pr1
c1       fetch sr0
```

```
# check for address match
c0 lw  0(pr0)   -> c2/cr0
c2 seq cr0, pr0 -> p
```

Figure 7-7: VP code for the `lookup` benchmark. The code uses a loop to traverse the nodes of a tree. A conditional fetch instruction exits the loop once the search is complete.

white. The code uses the Floyd-Steinberg dithering algorithm to diffuse quantization errors, and it handles white pixels as a special case since they do not contribute to the error.

Table 7.5 compares the performance of a conditional branch Scale mapping (Scale-branch) to one which uses predication instead (Scale-pred). In both mappings, the VP code to process a pixel comprises four AIBs. In Scale-branch, a vector-fetched AIB checks for a white pixel before conditionally fetching the second AIB; then, the fetches in the second and third AIBs are unconditional. In place of the VP fetch instructions, Scale-pred uses four vector-fetches to execute the AIBs. Thus, the VP fetch instructions add some overhead to Scale-branch, but it executes only 3 VP instructions for white pixels versus 30 for non-white pixels. Depending on the white content of the input image, the performance of Scale-branch can greatly exceed Scale-pred. For the reference input image, a gradation which has only 6.3% white pixels, Scale-branch runs around 2% slower than Scale-pred. However, for the dragon drawing with 64% white pixels, Scale-branch processes 289 pixels per thousand cycles compared to 211 for Scale-pred, a 37% increase.

7.3.3 Data-Parallel Loops with Inner-Loops

Often an inner loop has little or no available parallelism, but the outer loop iterations can run concurrently. For example, the EEMBC `lookup` benchmark models a router using a PATRICIA tree [Mor68] to perform IP route lookups. The benchmark searches the tree for each IP address in an input array, with each lookup chasing pointers through around 5–12 nodes of the tree. Very little parallelism is available in each search, but many searches can run simultaneously.

In the Scale implementation, each VP handles one IP lookup. The computation begins with a

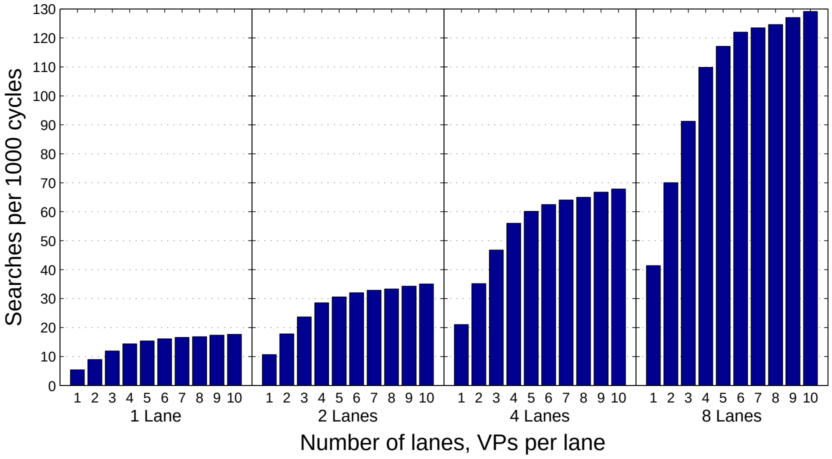


Figure 7-8: Performance of the `lookup` benchmark as a function of number of lanes and number of VPs per lane. The performance metric is the number of searches completed per 1000 clock cycles.

vector-load of the IP addresses, followed by a vector-fetch to launch the VP threads. The VP code, shown in Figure 7-7, traverses the tree using a loop composed of three AIBs. Although it only contains 10 VP instructions, one iteration of this inner loop takes around 21 cycles to execute on Scale due to the memory, fetch, and inter-cluster communication latencies. The loop is split into three AIBs to maximize the potential parallelism—after doing a load, a VP does not use the result until its next AIB, allowing the execution of other VPs to be interleaved with the memory access.

The microarchitectural simulator gives us the ability to evaluate a range of Scale configurations. To show scaling effects, we model Scale configurations with one, two, four, and eight lanes. For eight lanes, the memory system is doubled to eight cache banks and two 64-bit DRAM ports to appropriately match the increased compute bandwidth. The same binary code is used across all Scale configurations.

Figure 7-8 evaluates the performance of `lookup` on Scale configurations with different number of lanes, and with different vector lengths. The VP code uses 3 private registers, allowing Scale to support 10 VPs per lane. However, in the study, the vector length is artificially constrained to evaluate performance with 1–10 VPs per lane. The results show the benefit of exploiting the thread parallelism *spatially* across different lanes: performance increases roughly linearly with the number of lanes. The results also demonstrate the importance of exploiting thread parallelism *temporally* by multiplexing different VPs within a lane: performance increases by more than $3\times$ as more VPs are mapped to each lane. With 4 lanes and a full vector length (40), Scale executes 6.1 compute-ops per cycle, 1.2 of which are loads. Including the overhead of initiating the searches and checking the final results, the overall rate for searching the tree is 0.58 node traversals per cycle, or 1.7 cycles per traversal.

7.3.4 Loops with Cross-Iteration Dependencies

Many loops are non-vectorizable because they contain loop-carried data dependencies from one iteration to the next. Nevertheless, there may be ample loop parallelism available when there are operations in the loop which are not on the critical path of the cross-iteration dependency. The vector-thread architecture allows the parallelism to be exposed by making the cross-iteration (cross-VP) data transfers explicit. In contrast to software pipelining for a VLIW architecture, the vector-thread code need only schedule instructions locally in one loop iteration. As the code executes on a vector-thread machine, the dependencies between iterations resolve dynamically and the performance automatically adapts to the software critical path and the available hardware resources.

ADPCM Decode Example. Compression and decompression algorithms generally have dependencies from one coded element to the next. In the audio domain, analog signals are digitized in a process called pulse code modulation (PCM). A differential PCM encoding takes advantage of the high correlation between successive samples to compress the digital data as a sequence of differences instead of a sequence of samples. An adaptive differential PCM (ADPCM) encoding changes the step size of the digital quantization based on the signal itself—small steps are used when the signal amplitude is small, while large steps are used when the signal amplitude is large. The code in Figure 4-6 shows an example ADPCM decoder which converts an 8-bit input stream to a 16-bit output stream. In each iteration of the loop, a new input value (`delta`) is combined with the previous history to determine the step size for the reconstructed output (`valpred`). The `index` and `valpred` variables are accumulated from one iteration to the next.

The `adpcm_dec` benchmark in the Mediabench suite is similar to the example in Figure 4-6, except the 16-bit signal is compressed as a stream of 4-bit values. This slightly complicates the code since each byte of compressed input data contains two elements. Another difference is that `valpred` is computed with a series of shifts and adds instead of a multiply. The Scale mapping is similar to that in Figure 4-6, and each VP handles one output element. Instead of vector-loading the input data, each VP uses a load-byte instruction, and then it conditionally uses either the lower or upper 4-bits depending on if it is even or odd (the VPs each hold their index number in a private register). Overall, the Scale AIB for `adpcm_dec` uses 33 VP instructions, compared to 16 in the example in Figure 4-6. Cluster 0 uses only one private register and seven shared registers, resulting in a maximum vector length of 100.

Figure 7-9 shows the theoretical peak performance of `adpcm_dec`, under the assumption that a loop iteration can execute at a rate of one instruction per cycle. The software critical path is through the two loop-carried dependencies (`index` and `valpred`) which each take 5 cycles to compute (Figure 4-6 shows the 5 VP instructions used to compute the saturating accumulations). Thus, a new iteration can begin executing—and an iteration can complete—at an interval of 5 cycles, resulting in 33 instructions executed every 5 cycles, or 6.6 instructions per cycle. In order to achieve this throughput, 7 iterations must execute in parallel, a feat difficult to achieve through traditional software pipelining or loop unrolling. An alternative is to extract instruction-level parallelism within and across loop iterations, but this is difficult due to the sequential nature of the code. In particular, an iteration includes 3 load-use dependencies: the input data, the `indexTable` value, and the `stepsizeTable` value. The minimum execution latency for an iteration is 29 cycles on Scale, even though the 33 VP instructions are parallelized across 4 clusters.

Scale exploits the inherent loop-level parallelism in `adpcm_dec` by executing iterations in parallel across lanes and by overlapping the execution of different iterations within a lane. Figure 7-10 diagrams a portion of the execution. Although the code within an iteration is largely serial, the Scale AIB partitions the instructions to take advantage of cluster decoupling. Cluster 0 loads the `delta`, `indexTable`, and `stepsizeTable` values, c1 accumulates `index` from the previous iteration to the next, c2 performs the `valpred` calculation, and c3 accumulates `valpred` from the previous iteration to the next. The cluster execution unfolds dynamically subject to these dependencies; i.e., c0 runs ahead and c3 trails behind. Across lanes, the explicit nextVP sends and prevVP receives allow the execution to dynamically adapt to the software critical path. The two cross-iteration dependencies propagate in parallel on c1 and c3. The combination of cluster decoupling within a lane and parallel execution across lanes allows up to 9 VPs (loop iterations) to execute concurrently in

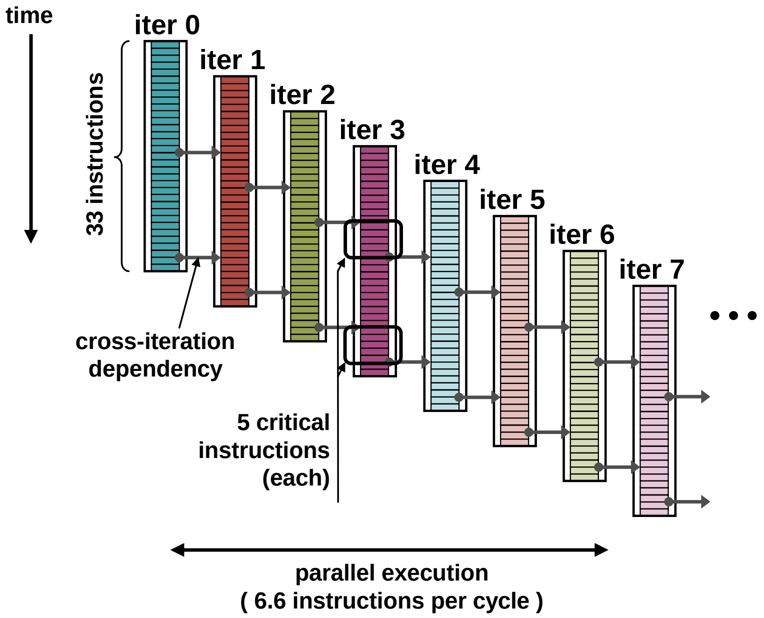


Figure 7-9: Ideal execution of the `adpcm_dec` benchmark, assuming each iteration executes one instruction per cycle. The two cross-iteration dependencies limit the parallel execution to one iteration per 5 cycles, or 6.6 instructions per cycle.

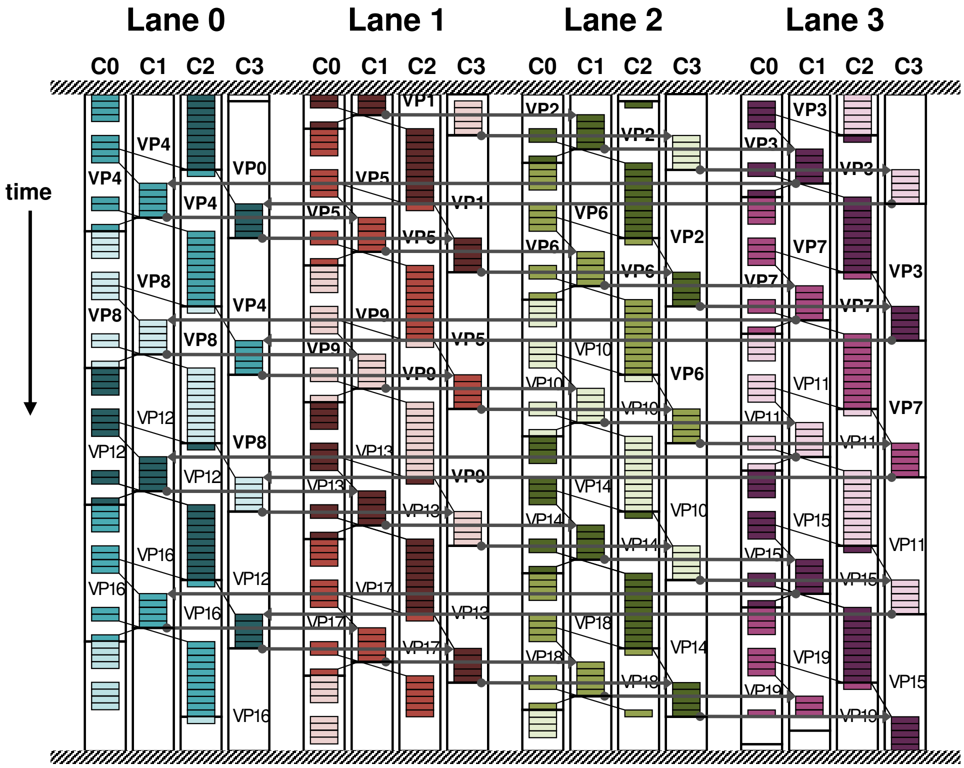


Figure 7-10: Execution of the `adpcm_dec` benchmark on Scale. Each small shaded box represents an individual compute-op. The narrow lines within a lane indicate inter-cluster data transfers, and the thicker arrows between lanes indicate cross-VP data transfers.

```
spin_on_lock:
  .aib begin
  c0 lw.atomic.or 0(sr0), 1 -> pr0 # try to get the lock (set to 1).
  c0 seq pr0, 0 -> c1/p            # did we get the lock (old value was 0)?
  c1  (p) addu.fetch pr0, (critical_section - spin_on_lock) -> pr0
  c1 (!p) addu.fetch pr0, (spin_on_lock - spin_on_lock) -> pr0
  .aib end
```

Figure 7-11: VP code for acquiring a lock. The lock address is held in c0/sr0 and the address of the current AIB is held in c1/pr1. If it acquires the lock, the VP fetches the "critical_section" AIB, otherwise it re-fetches the same AIB to try again.

the VTU. The output data is written to the store-data cluster and then transfered to memory using vector-stores which operate in the background of the computation.

Including all overheads, Scale achieves 97.5% of its ideal performance for `adpcm_dec`. It executes 6.46 compute-ops per cycle and completes a loop iteration every 5.13 cycles. The input stream is processed at 0.20 bytes per cycle, and the output rate is 0.78 bytes per cycle. This is a speedup of 7.6× compared to running the ADPCM decode algorithm on the control processor alone.

Other Examples. The two MiBench cryptographic kernels, `sha` and `rijndael`, have many loop-carried dependences. The `sha` mapping uses 5 cross-VP data transfers, while the `rijndael` mapping vectorizes a short four-iteration inner loop. Scale is able to exploit instruction-level parallelism within each iteration of these kernels by using multiple clusters, while also exploiting limited cross-iteration parallelism on multiple lanes.

7.3.5 Reductions and Data-Dependent Loop Exit Conditions

Scale provides efficient support for arbitrary reduction operations by using shared registers to accumulate partial reduction results from multiple VPs on each lane. The shared registers are then combined across all lanes at the end of the loop using the cross-VP network. The `pktflow` code uses reductions to count the number of packets processed.

Loops with data-dependent exit conditions ("while" loops) are difficult to parallelize because the number of iterations is not known in advance. For example, the `strcmp` and `strcpy` standard C library routines, used in the `text` benchmark, loop until the string termination character is seen. The cross-VP network can be used to communicate exit status across VPs but this serializes execution. Alternatively, iterations can be executed speculatively in parallel and then nullified after the correct exit iteration is determined. The check to find the exit condition is coded as a cross-iteration reduction operation. The `text` benchmark executes most of its code on the control processor, but uses this technique for the string routines to attain a 1.5× overall speedup.

7.3.6 Free-Running Threads

When structured loop parallelism is not available, VPs can be used to exploit arbitrary thread parallelism. With free-running threads, the control processor interaction is minimized. Each VP thread runs in a continuous loop processing tasks from a shared work-queue (in memory). These tasks may be queued by the control processor or the VPs themselves. The VP threads use atomic memory operations to synchronize accesses to shared data structures. Figure 7-11 shows a simple example of an AIB to acquire a lock.

```
vp_reverse_strcmp:
    .aib begin
    c0 lbu 0(pr1)          -> c1/cr0, c2/cr0 # get byte from 2nd string
    c1 sne cr0, 0          -> p              # is the byte 0?
    c0 lbu 0(pr0)          -> c1/cr1, c2/cr1 # get byte from 1st string
    c1 (p) seq cr0, cr1    -> p              # determine whether to continue
    c2 subu cr0, cr1       -> c1/pr2         # calculate the return value
    c1 psel.fetch pr1, pr0  -> pr0           # either loop or return
    c0 addu pr0, 1         -> pr0            # increment the 1st pointer
    c0 addu pr1, 1         -> pr1            # increment the 2nd pointer
    .aib end
```

Figure 7-12: VP function for string comparison. Pointers to the two character arrays are passed in c0/pr0 and c0/pr1. The current AIB address is held in c1/pr0, and the return address is passed in c1/pr1. The result is returned in c1/pr2. The function executes as a loop, repeatedly fetching the same AIB as long as the string characters are identical. It returns to the caller once a difference is found.

The free-running threads method can also be used as an alternative to vector-fetched VP threads, even when structured parallelism is available. Since the VPs schedule their own work, they can run continuously to keep the VTU constantly utilized. In comparison, when VP threads are repeatedly launched by the control processor, the lane command management units must wait until one set finishes before initiating the next (this is described in Section 5.3.2). This is generally not a problem if the VP threads all run for the same duration, but the lane can be underutilized when one VP runs for longer than the others. Free-running threads can improve the load-balancing by eliminating unnecessary synchronization points.

The quick-sort algorithm (`qsort`) is an example of a kernel implemented on Scale using free-running threads. The divide-and-conquer algorithm recursively partitions an input list into sublists which are sorted independently. The potential parallelism grows with each recursion. The Scale mapping includes a shared work queue of lists to be sorted. A VP thread 1) pops a task from the head of the work queue, 2) partitions the target list into two sublists, 3) pushes one of the sublists onto the tail of the work queue, and then 4) proceeds to process the other sublist. When a VP thread reaches the end of the recursion and its list does not divide into sublists, it repeats the process and checks the work queue to handle any remaining tasks. The VP threads terminate when the work queue is empty, at which point the sorting is complete.

In comparison to the simple VP threads described in previous examples, those for `qsort` are significantly more substantial. The VP code includes around 450 VP instructions organized into 65 AIBs with relatively complicated control flow. The code also uses more private registers than the other examples, and the vector length is limited to 12 (3 VPs per lane).

The quick-sort algorithm provided by the C Standard Library is a polymorphic function that can sort different types of objects. To achieve this, a pointer to the comparison routine used to make sorting decisions is passed as an argument to `qsort`. The Scale implementation retains this feature, and the VPs actually execute a call to a separately defined function. An example is the string comparison function shown in Figure 7-12. The calling convention passes arguments in private VP registers, and it includes a return address. The register configuration and usage policies would have to be specified in much more detail in a more rigorously defined application binary interface (ABI) for VPs.

Quick-sort's free-running threads stress Scale in different ways than benchmarks with more structured parallelism. The VP thread fetch instructions have an AIB cache miss rate of 8.2%. However, this should not be interpreted in the same way as the instruction cache miss rate on a scalar architecture. Since instructions are grouped together in AIBs, the miss rate per compute-op is actually only 0.99%, with an average of 101 compute-ops executed per miss. Scale's decoupled AIB fill unit and time-multiplexing of VP threads are able to effectively hide much of the latency of these AIB cache misses: even when the benchmark is run in a "magic AIB cache fill" mode, the performance only improves by 6%.

Despite having a high cache miss rate, `qsort` executes an average of 1.9 compute-ops per cycle and achieves a 3× speedup compared to the control processor alone. With a magic main memory that has a zero-cycle latency and unlimited bandwidth, performance improves by 57%.

7.3.7 Mixed Parallelism

Some codes exploit a mixture of parallelism types to accelerate performance and improve efficiency. The garbage collection portion of the lisp interpreter (`li`) is split into two phases: mark, which traverses a tree of currently live lisp nodes and sets a flag bit in every visited node, and sweep, which scans through the array of nodes and returns a linked list containing all of the unmarked nodes [Asa98]. During mark, the Scale code sets up a queue of nodes to be processed and uses a stripmine loop to examine the nodes, mark them, and enqueue their children. In the sweep phase, VPs are assigned segments of the allocation array, and then they each construct a list of unmarked nodes within their segment in parallel. Once the VP threads terminate, the control processor vector-fetches an AIB that stitches the individual lists together using cross-VP data transfers, thus producing the intended structure. Although the garbage collector has a high cache miss rate, the high degree of parallelism exposed in this way allows Scale to sustain 2.7 operations/cycle and attain a speedup of 5.3 over the control processor alone.

7.4 Locality and Efficiency

The strength of the Scale VT architecture is its ability to capture a wide variety of application parallelism while using simple microarchitectural mechanisms that exploit locality. This section evaluates the effectiveness of these mechanisms, and Table 7.6 presents a variety of statistics for each benchmark. The data characterizes the VP configuration and vector-length, and the use of Scale's control and data hierarchies.

7.4.1 VP Configuration

Shared registers serve two purposes on Scale, they can hold shared constants used by all VPs and they can be used as temporary registers within an AIB. The use of shared registers increases the virtual processor vector-length, or equivalently it reduces the register file size needed to support a large number of VPs. Table 7.6 shows that more often than not the VPs are configured with more shared than private registers. The significant variability in VP register configurations for different kernels highlights the importance of allowing software to configure VPs with just enough of each register type.

A moderate vector length allows the control processor overhead to be amortized by many VPs, and it provides the VTU with parallelism to hide functional unit and cache access latencies. Longer vector lengths also improve the efficiency of vector memory commands. As shown in the table, around $2/3$ of the benchmarks sustain average vector-lengths of 16 or more. 9 out of the 32 benchmarks have an average vector length greater than 50. A few benchmarks—like `rotate`, `fdct`, `idct`, and `scrambler`—have vector lengths constrained by large private register counts. Others have shorter vector lengths due to parallelism limitations inherent to the algorithms.

					Averages Per 100 VTU Compute-Ops																				
	VP config. (Averages)				Control Hierarchy						Data Hierarchy														
											Sources			Dests.				Loads				Stores			
Benchmark	Private Regs	Shared Regs	VL Max	Vector Length	CP Instructions	Vector-Fetches	Thread-Fetches	Fetch EDs	AIBs	AIB$ Misses	Chain Registers	Register File	Immediates	Chain Registers	Register File	Lane Transports	NextVP Transports	Elements	Addresses	Cache Bytes	DRAM Bytes	Elements	Addresses	Cache Bytes	DRAM Bytes
rgbcmy	5	5	40	40	1.3	0.3	–	1.2	11.8	–	65	118	18	53	76	76	–	18	7	18	18	6	2	24	26
rgbyiq	6	16	28	28	1.4	0.2	–	0.7	4.8	–	43	157	14	43	71	71	–	14	6	14	14	14	5	14	31
hpg	5	3	64	64	1.7	0.4	–	1.7	27.3	–	55	136	9	55	82	82	–	27	8	27	9	9	2	9	21
text	4	4	85	7	318.7	2.3	–	9.3	15.6	–	24	74	12	24	68	52	15	24	9	24	–	4	1	4	1
dither	10	12	36	30	5.1	0.1	10	10.3	13.4	–	36	113	31	39	53	50	–	23	16	37	4	5	3	9	–
rotate	22	0	16	12	2.9	0.5	–	1.9	5.9	–	0	79	53	0	58	37	–	5	2	5	–	5	5	5	–
lookup	8	8	40	40	0.2	0.0	30	29.8	30.8	–	1	122	20	1	49	30	–	21	20	84	–	0	0	0	–
ospf	4	1	28	28	0.2	0.0	6	5.8	5.8	0.9	9	110	46	15	77	48	–	20	20	78	–	9	8	37	–
pktflow (512KB)	5	12	56	46	0.9	0.4	3	4.6	23.1	–	55	80	41	54	51	39	–	20	19	49	–	2	0	6	–
pktflow (1MB)	5	12	56	51	0.7	0.4	3	4.5	23.1	–	55	80	42	54	51	38	–	20	19	49	106	2	0	6	6
pktflow (2MB)	5	12	56	52	0.6	0.4	3	4.6	23.1	–	55	80	42	54	51	38	–	20	19	49	105	2	0	6	6
pntrch	6	16	70	13	0.0	0.0	14	14.2	14.2	0.2	29	107	36	36	21	29	–	14	14	57	–	0	0	1	–
fir-hl	5	4	64	60	4.3	0.4	–	1.7	25.3	–	49	148	3	49	78	49	–	25	11	99	5	1	1	5	3
fbital (step)	4	9	116	20	10.6	0.5	10	12.5	20.8	–	12	114	26	12	78	73	11	16	10	32	–	1	0	1	–
fbital (pent)	4	9	116	100	1.6	0.1	8	8.1	15.5	–	15	124	23	15	84	77	8	15	10	31	–	0	0	1	–
fbital (typ)	4	9	116	85	1.2	0.1	8	8.2	15.7	–	15	130	23	15	84	77	8	15	10	31	–	7	2	14	–
fft	11	1	32	26	14.3	1.3	–	5.2	33.0	–	33	186	17	33	112	67	–	45	22	91	–	37	9	74	–
viterb	16	10	24	16	8.2	1.2	4	9.1	23.0	–	31	153	16	31	72	72	–	10	4	40	–	11	5	42	–
autocor (data1)	3	2	124	8	23.5	4.5	–	17.8	35.6	–	63	131	4	63	84	66	–	36	18	71	–	2	0	4	–
autocor (data2)	3	2	124	16	8.4	2.1	–	8.3	33.4	–	67	133	0	67	75	67	–	35	17	71	–	0	0	0	–
autocor (data3)	3	2	124	32	4.2	1.0	–	4.2	33.4	–	67	133	0	67	71	67	–	34	14	69	–	0	0	0	–
conven (data1)	5	7	60	32	3.5	0.4	–	1.6	12.5	–	25	117	58	33	108	46	–	8	6	17	–	2	1	8	–
conven (data2)	5	7	60	32	3.5	0.4	–	1.6	12.5	–	30	115	55	37	105	48	–	10	7	20	–	2	1	10	–
conven (data3)	5	7	60	32	3.9	0.4	–	1.7	13.3	–	33	113	53	40	100	47	–	13	9	27	–	3	1	13	–
rijndael	13	8	24	4	11.1	1.9	–	7.7	7.7	–	77	94	27	76	44	50	–	33	32	132	–	4	2	15	–
sha	1	4	114	6	19.0	1.2	–	3.7	6.7	–	103	30	41	89	17	54	19	17	14	68	–	5	3	18	–
qsort	25	19	12	12	26.2	0.0	12	12.0	12.0	1.0	9	127	71	8	63	5	–	20	20	70	68	17	17	67	–
adpcm_enc	3	20	96	91	0.3	0.0	–	0.1	2.0	–	26	79	26	24	48	44	8	6	5	20	–	1	1	1	–
adpcm_dec	1	16	100	100	0.5	0.0	–	0.1	3.0	–	18	107	38	18	46	12	9	9	9	27	–	3	1	6	–
li.gc	4	5	113	63	0.7	0.1	8	8.2	11.3	–	34	73	68	35	24	50	2	12	11	47	55	9	9	34	10
rgbycc (1k)	6	10	32	31	1.5	0.2	–	0.6	4.8	–	86	114	14	86	29	71	–	14	6	14	–	14	4	14	–
rgbycc (100k)	6	10	32	32	1.3	0.1	–	0.6	4.8	–	86	114	14	86	29	71	–	14	6	14	14	14	4	14	35
fdct (32)	16	38	12	9	4.3	0.4	–	1.4	3.4	–	59	149	4	57	70	61	–	14	6	27	–	14	4	27	–
fdct (1000)	16	38	12	10	4.1	0.4	–	1.4	3.4	–	59	149	4	57	70	61	–	14	7	27	28	14	4	27	17
idct (32)	18	28	12	9	3.9	0.3	–	1.3	3.1	–	72	128	3	64	56	61	–	13	6	25	–	12	4	25	–
idct (1000)	18	28	12	10	3.8	0.3	–	1.3	3.1	–	72	128	3	64	56	61	–	13	6	25	25	12	4	25	17
quantize (32)	3	6	120	114	0.9	0.1	–	0.3	8.3	–	8	108	25	8	100	75	–	25	8	42	–	8	2	17	–
quantize (1000)	3	6	120	120	0.8	0.1	–	0.3	8.3	–	8	108	25	8	100	75	–	25	8	42	25	8	2	17	34
jpegenc	16	26	16	4	16.8	1.2	6	10.8	10.8	–	23	71	34	23	41	40	17	12	11	21	1	1	1	1	–
scrambler (4k)	8	4	12	12	6.6	0.5	–	1.9	5.7	0.2	23	184	6	23	99	22	–	22	14	87	–	22	11	86	–
scrambler (100k)	8	4	12	12	6.2	0.5	–	1.9	5.7	–	23	187	6	23	100	23	–	23	15	90	91	23	11	90	118
convolenc (4k)	4	4	56	54	0.8	0.1	–	0.4	6.1	–	73	79	48	73	33	36	–	15	13	27	–	3	1	12	–
convolenc (100k)	4	4	56	56	0.7	0.1	–	0.4	6.1	–	73	79	48	73	33	36	–	15	13	27	6	3	1	12	13
interleaver (4.8k)	9	8	36	33	0.3	0.1	–	0.5	3.9	–	83	79	42	83	18	58	–	1	1	4	–	4	1	4	–
interleaver (96k)	9	8	36	36	0.2	0.1	–	0.4	3.9	–	83	79	42	83	18	58	–	1	1	4	4	4	1	4	23
mapper (1.2k)	8	2	20	17	3.4	0.5	–	2.2	9.0	–	58	127	42	58	50	71	–	36	39	130	–	38	16	153	–
mapper (48k)	8	2	20	20	2.5	0.4	–	1.8	9.0	–	58	127	43	58	50	71	–	37	41	131	15	38	16	153	249
ifft (32)	19	10	16	16	1.1	0.2	–	1.0	3.9	1.0	64	86	43	58	41	48	–	4	3	16	–	3	2	14	–
ifft (1000)	19	10	16	16	1.0	0.2	–	1.0	3.9	1.0	64	86	43	58	40	48	–	4	3	16	24	3	2	14	4

Table 7.6: VP configuration, control hierarchy, and data hierarchy statistics for Scale benchmarks. The VP configuration statistics are averages of data recorded at each vector-fetch command. The private and shared register counts represent totals across all four clusters. The control and data hierarchy statistics are presented as averages per 100 executed compute-ops. The source and destination operand stats include vector-load writebacks and vector-store reads. The load addresses statistic includes accesses by the vector refill unit.

7.4.2 Control Hierarchy

A VT machine amortizes control overhead by exploiting the locality exposed by AIBs and vector-fetch commands, and by factoring out common control code to run on the control processor. A vector-fetch broadcasts an AIB address to all lanes and each lane performs a single tag-check to determine if the AIB is cached. On a hit, an execute directive is sent to the clusters which then retrieve the instructions within the AIB using a short (5-bit) index into the small AIB cache. The cost of each instruction fetch is on par with a register file read. On an AIB miss, a vector-fetch will broadcast AIBs to refill all lanes simultaneously. The vector-fetch ensures an AIB will be reused by each VP in a lane before any eviction is possible. When an AIB contains only a single instruction on a cluster, a vector-fetch will keep the ALU control lines fixed while each VP executes its operation, further reducing control energy.

As an example of amortizing control overhead, `rbgycc` runs on Scale with a vector-length of 32 and vector-fetches an AIB with 21 VP instructions (Section 7.3.1). Thus, each vector-fetch executes 672 compute-ops on the VTU, 168 per AIB cache tag-check in each lane. In Table 7.6, this data appears as 0.1 vector-fetches and 0.6 fetch EDs per 100 compute-ops. Also, since each AIB contains 21 compute-ops, there are 4.8 AIBs per 100 compute-ops ($1/21 \times 100$). As shown in the table, most of the benchmarks execute more than 100 compute-ops per vector-fetch, even non-vectorizable kernels like `adpcm_dec/_enc`. The execute directive and AIB cache tag-check overhead occurs independently in each lane, but there are still generally only a few fetch EDs per 100 compute-ops. The kernels with more thread-fetches have greater control overhead, but it is still amortized over the VP instructions in an AIB. Even `qsort`—which only uses thread-fetches—generates only 12 fetch EDs per 100 compute-ops. A few of the benchmarks average 25–33+ AIBs per 100 compute-ops (3 or 4 compute-ops per AIB), but most have many compute-ops grouped into each AIB and thus far lower AIB overheads.

7.4.3 Data Hierarchy

AIBs help in the data hierarchy by allowing the use of chain registers to reduces register file energy. Table 7.6 shows the prevalence of chain registers for compute-op sources and destinations. 19 out of 32 benchmarks use fewer than 1.2 register file operands per compute-op. Half of the benchmarks have at least 40% of their compute-ops targeting chain registers.

Clustering in Scale is area and energy efficient and cluster decoupling improves performance. The clusters each contain only a subset of all possible functional units and a small register file with few ports, reducing size and energy. Each cluster executes compute-ops and inter-cluster transport operations in order, requiring only simple interlock logic with no inter-thread arbitration or dynamic inter-cluster bypass detection. Independent control on each cluster enables decoupled cluster execution to hide inter-cluster or cache access latencies. Table 7.6 shows that $3/4$ (24) of the benchmarks have at least 40% of their compute-ops targeting remote clusters. 10 of the benchmarks have more than 70% of their compute-ops targeting remote clusters.

Vector memory commands enforce spatial locality by moving data between memory and the VP registers in groups. This improves performance and saves memory system energy by avoiding the additional arbitration, tag-checks, and bank conflicts that would occur if each VP requested elements individually. Table 7.6 shows the reduction in memory addresses from vector memory commands. The maximum improvement is a factor of four, when each unit-stride cache access loads or stores one element per lane, or each segment cache access loads or stores four elements per lane. 13 benchmarks load two or more elements per cache access, and 20 benchmarks store two or more elements per access. Scale can exploit memory data-parallelism even in loops with non-data-parallel compute. For example, the fbital, text, and adpcm benchmarks use vector memory commands to access data for vector-fetched AIBs with cross-VP dependencies.

Table 7.6 shows that the Scale data cache is effective at reducing DRAM bandwidth for many of the benchmarks. For example, hpg retrieves 3 bytes from the cache for every byte from DRAM, dither and mapper read around 9× more cache bytes than DRAM bytes, and for fir-hl the ratio is about 20×. A few exceptions are pktflow, li, and idct, for which the DRAM bytes transferred exceed the total bytes accessed. Scale always transfers 32-byte lines on misses, but support for non-allocating loads could potentially help reduce the bandwidth in these cases. For stores, fir-hl, li, fdct, idct, and ifft have cache bandwidths that are 1.5–3.5× greater than the DRAM bandwidth. However, several benchmarks have a DRAM store bandwidth that is greater than their cache store bandwidth. This is usually due to insufficient element merging within the 16-byte non-allocating store blocks. In some cases, merging increases if the memory output port becomes a bottleneck. This is because the data blocks remain on-chip for a longer amount of time, increasing the opportunity to capture the spatial locality.

7.5 Applications

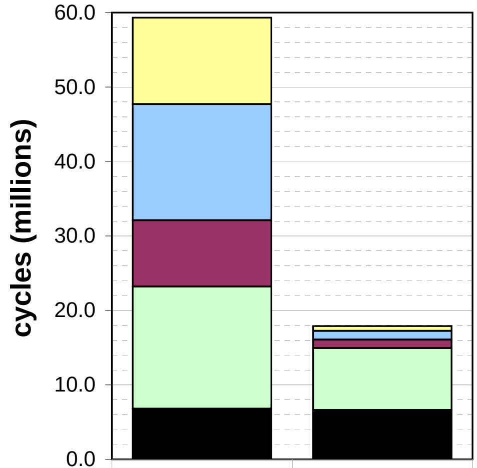
This section presents performance results for larger applications mapped to Scale. Both examples come from the XBS benchmark suite [Ger05], and they each combine several kernels from Section 7.2.

JPEG Encoder. The JPEG encoder application compresses red-green-blue (RGB) input pixels into a JPEG-formatted image [PM93]. The implementation used here is based on the work done by the Independent JPEG Group [The]. The encoder first converts the RGB pixels into the YCbCr color space using the rgbycc kernel. The pixels are than processed as 8×8 blocks. First they are transformed into a frequency domain representation using the fdct kernel. Then the blocks are quantized, with greater emphasis given to the lower frequency components. The quantization routine for Scale optimizes the quantize kernel used in Section 7.2 by converting divide operations into multiplies by a reciprocal [Ger05]. Finally, the jpegenc kernel compresses blocks using differential pulse code modulation, run-length encoding, and Huffman coding.

Figure 7-13 compares the performance of the compiled code running on the RISC control processor to the hand-optimized Scale code which uses the VTU. The vectorizable portions of the application (rgbycc, fdct, and quantize) have very large speedups. Most of the remaining runtime in the Scale mapping comes from jpegenc and the non-parallelized application overhead code. Overall, Scale encodes the image 3.3× faster than the RISC baseline. Scale averages 68.2 cycles per pixel, 3.8 million pixels per second at 260 MHz.

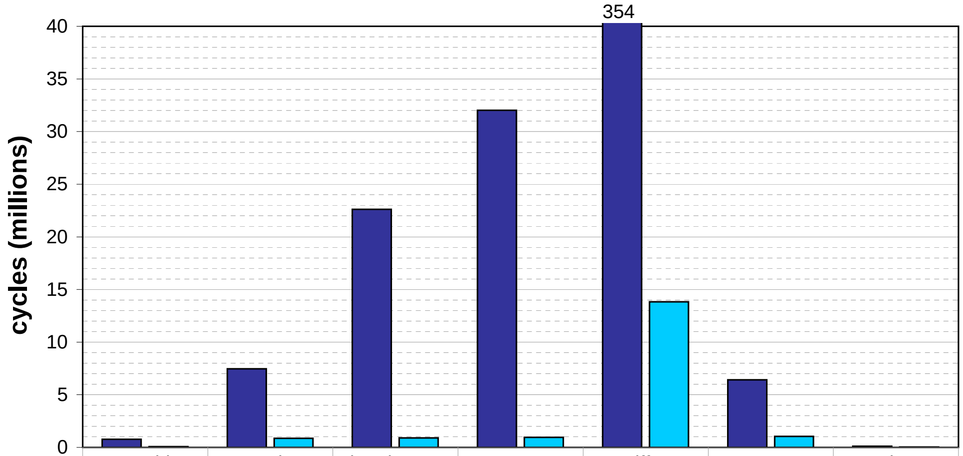
802.11a Wireless Transmitter. The IEEE 802.11a transmitter application prepares data for wireless communication [IEE99]. It does this by transforming binary input data into orthogonal frequency division multiplexing (OFDM) symbols. The XBS implementation supports the 24 Mbps 802.11a data rate. The translation is performed by streaming data through a sequence of kernels: scrambler, convolenc, interleaver, mapper, ifft, and cycext.

Figure 7-14 compares the performance of optimized compiled C code running on the RISC control processor to the hand-optimized Scale implementation which uses the VTU. The ifft runtime dominates the performance of both mappings. Scale achieves large speedups on this vectorizable application, and the overall performance improvement over RISC is 24×. Over the entire run, Scale executes an average of 9.7 VTU compute operations per cycle while also fetching 0.55 load elements and writing 0.50 store elements per cycle. The off-chip memory bandwidth is 0.14 load bytes



	RISC	Scale
🟨 rgbycc	11.6	0.7
🟦 fdct	15.6	1.1
🟥 quantize	8.9	1.2
🟩 jpegenc	16.4	8.3
■ other	6.8	6.6

Figure 7-13: JPEG encoder performance. The input is a 512×512 pixel image (0.75 MB).



	scrambler	convolenc	interleaver	mapper	ifft	cycext	other
🟦 RISC	0.76	7.47	22.61	32.03	354.02	6.42	0.10
🟦 Scale	0.07	0.87	0.89	0.95	13.82	1.04	0.03

Figure 7-14: IEEE 802.11a transmitter performance. The input is 240 kB. The total cycle counts are 423.4 million for RISC and 17.7 million for Scale.

and 0.72 store bytes per cycle. The Scale 802.11a transmitter processes input data at a rate of 74 cycles per byte, 28.2 Mbps at 260 MHz.

7.6 Measured Chip Results

This section presents performance, power, and energy measurements for benchmarks running on the Scale chip (see Tables 7.7 and 7.8). Due to the memory bandwidth constraints imposed by the system test infrastructure (explained in Section 6.1.7), results are only reported for benchmarks with data sets that fit within Scale's 32 KB cache. However, since the benchmarks include additional code that is outside of the timing loop, the Scale chip is run in its caching enabled mode of operation. The performance results are reported as per-cycle throughputs which are independent of operating frequency. The power measurements were made at an operating frequency of 160 MHz. Figure 6-15 shows that power scale's linearly with frequency, and there is a factor 1.6 increase from 160 MHz to 260 MHz. In addition to performance throughputs, the tables show energy throughputs for the benchmarks. These are also based on the 160 MHz operating frequency, but Figure 6-16 shows that at a given supply voltage (1.8 V in this case) energy consumption does not vary significantly with frequency.

The result tables show performance speedups for the assembly-optimized code running on Scale compared to the compiled code running on the control processor. Similar caveats apply to this comparison as were mentioned in Section 7.2. In particular, the Scale code is hand-tuned and the control processor does not have an analogue to Scale's 16-bit multiply instruction. The result tables also compare the CP results to the idealistic RISC simulations from Section 7.2. The control processor runs 7–150% slower, with many benchmarks falling within the 25–60% range. The primary reasons for these slowdowns are (1) CP instruction fetch stalls after misses in the L0 buffer or branch mispredictions, (2) Scale's 2-cycle load-use latency for cache hits, and (3) the latency of the control processor's iterative multiply/divide unit. The result tables show that the measured results for code running on the VTU are generally in line with simulation results. 13 out of the 21 benchmarks have a simulation error of less than 5%, and 17 have an error less than 10%. The outliers are generally benchmarks which execute more code on the control processor (see *CP Instructions* in Table 7.6). For example, `text` executes 3.2× more CP instructions than compute-ops, `viterb` executes 8 CP instructions per 100 compute-ops, and `jpegenc` executes 17 CP instructions per 100 compute-ops. The accuracy for `fbital`, `autocor`, and `conven` improves for runs with less control processor overhead. Overall, the measured chip performance validates the simulation results used elsewhere in this thesis. The measured speedups on Scale are generally significantly higher than those reported in Section 7.2 due to the ideal RISC model used in the simulator.

The results in the table show a significant variance in Scale's power consumption across different benchmarks. The range is from 242 mW for `jpegenc` to 684 mW for `idct`, a factor of 2.8×. At 260 MHz, these would scale to around 400 mW and 1.1 W respectively. The variation in power consumption corresponds well with the benchmark utilization rates, i.e. the per-cycle throughputs for compute operations, load elements, and store elements which are shown in the result tables in Section 7.2. For example, `idct` has the highest utilization: it executes 11.3 compute-ops per cycle while reading 1.4 load elements and writing 1.4 store elements. `jpegenc` only averages 1.7 compute-ops and 0.2 load elements per cycle. The spread in power consumption across the benchmarks shows the effectiveness of clock-gating in the Scale chip. This gating is directly enabled by the Scale VT architecture's partitioning of work across decoupled clusters and vector memory units: when blocks are inactive, they do not consume power.

As a comparison point, Tables 7.7 and 7.8 also show the power and energy for the compiled

Benchmark	Input	Iterations per million cycles		Speed-up	Average power (mW)		Iterations per millijoule		Energy Ratio	Simulator performance difference %	
		CP	Scale		CP	Scale	CP	Scale		RISC	Scale
text	323 rule strings (14.0 KB)	0.6	0.8	1.4	73	249	1.3	0.5	0.4	33.5	43.0
rotate	12.5 KB image	1.4	42.8	31.7	58	449	3.7	15.3	4.1	48.8	2.9
lookup	2000 searches	3.2	32.1	10.2	45	396	11.2	13.0	1.2	47.5	5.4
ospf	400 nodes × 4 arcs	10.3	16.2	1.6	67	243	24.7	10.7	0.4	82.8	9.7
pktflow	512 KB (11.7 KB headers)	11.8	297.5	25.3	67	394	28.0	120.8	4.3	43.4	11.8
pntrch	5 searches, ≈ 350 nodes each	14.1	98.3	7.0	72	266	31.2	59.1	1.9	75.9	-2.0
fbital	step (20 carriers)	20.8	707.2	34.0	61	274	54.4	412.8	7.6	57.4	6.8
	pent (100 carriers)	2.2	160.8	73.3	58	316	6.1	81.4	13.4	54.2	1.4
	typ (256 carriers)	1.5	60.1	41.4	58	335	4.0	28.7	7.2	53.8	0.6
fft	*all* (1 KB)	7.8	289.9	37.3	42	455	29.4	101.9	3.5	121.4	4.3
viterb	*all* (688 B)	3.2	36.2	11.4	73	401	6.9	14.4	2.1	24.2	11.6
autocor	data1 (16 samples, 8 lags)	316.1	8264.2	26.1	19	335	2676.2	3949.4	1.5	123.6	15.9
	data2 (1024 samples, 16 lags)	2.0	161.3	78.8	21	503	15.6	51.3	3.3	130.6	-0.0
	data3 (500 samples, 32 lags)	2.1	197.6	92.0	21	475	16.3	66.5	4.1	130.5	0.2
conven	data1 (512 B, 5 xors/bit)	6.0	7299.0	1207.9	47	500	20.7	2337.2	113.1	26.1	-1.8
	data2 (512 B, 5 xors/bit)	7.0	8849.1	1265.0	49	518	22.9	2731.2	119.5	26.7	-0.4
	data3 (512 B, 3 xors/bit)	8.7	11235.3	1295.4	50	524	27.9	3431.9	122.9	29.2	-6.7

Table 7.7: Measured Scale chip results for EEMBC benchmarks. The *CP* columns show performance and power consumption for compiled out-of-the-box code (OTB) running on the Scale control processor. The *Scale* columns show results for assembly-optimized code (OPT) which makes use of the VTU. The power measurements were made with the chip running at 160 MHz. The chip idle power (169 mW) was subtracted from the *CP* power and energy measurements, but the *Scale* measurements report the total chip power consumption. The right-most columns compare the simulation results (Section 7.2) to the measured chip results, reporting the positive or negative percentage performance difference (simulator versus chip). The idealistic control processor simulator model (*RISC*) is compared to the actual Scale control processor.

Benchmark	Input	Input elements per 1000 cycles		Speed-up	Average power (mW)		Input elements per microjoule		Energy Ratio	Simulator performance difference %	
		CP	Scale		CP	Scale	CP	Scale		RISC	Scale
rgbycc	1 k pixels (2.9 KB)	10.30	431.31	41.9	53	536	31.1	128.7	4.1	8.4	0.7
fdct	32 8×8 blocks (4 KB)	0.37	11.41	31.0	61	670	1.0	2.7	2.8	6.9	0.5
idct	32 8×8 blocks (4 KB)	0.30	11.06	36.3	64	684	0.8	2.6	3.4	10.0	-0.5
quantize	32 8×8 blocks (4 KB)	0.24	1.67	7.0	41	249	0.9	1.1	1.2	148.2	5.4
jpegenc	1 8×8 block (128 B)	0.31	1.07	3.5	89	242	0.6	0.7	1.3	51.7	20.2
scrambler	4 k bytes	407.43	2989.43	7.3	83	384	787.3	1244.6	1.6	17.2	3.3
convolenc	4 k bytes	27.58	269.84	9.8	69	348	64.2	124.0	1.9	20.8	2.4
interleaver	4.8 k bytes	20.33	536.75	26.4	89	415	36.5	207.0	5.7	9.6	0.7
mapper	1.2 k bytes	9.86	502.12	50.9	63	423	25.2	189.9	7.5	55.0	8.5
ifft	32 64-pt blocks (8 KB)	0.03	1.47	47.4	60	554	0.1	0.4	5.2	87.0	-1.1

Table 7.8: Measured Scale chip results for XBS benchmarks. The performance and power results are based on input elements (pixels, blocks, or bytes) rather than iterations. Other columns are as described in Table 7.7.

benchmarks running on the control processor. The clocking overhead for the VTU is mitigated by subtracting out the chip idle power (169 mW). This is conservative since it also eliminates the idle clocking overhead for the control processor itself. However, the presence of the VTU also adds overhead to the control processor's cache accesses, and this overhead is not accounted for. Still, the reported CP power consumption is generally in line with the 56 mW at 160 MHz that we measured for a test-chip with a simple RISC core directly connected to 8 KB of RAM [BKA07]. The control processor generally uses around 45–75 mW. `autocor` uses much less power because the CP datapath is often stalled waiting for the iterative multiplier.

Scale's power draw is generally 3–10× greater than the control processor alone. However, the energy throughput data in the result tables reveals that the extra power is put to good use. In addition to achieving large performance speedups, the VTU code generally consumes significantly less energy than the CP code. Scale processes over twice as much work per joule compared to the control processor alone for 13 out of the 21 benchmarks, and on many of the vectorizable codes its efficiency improvement is over 4×. Threaded codes like `lookup` and those with low parallelism like `jpegenc` are less efficient, but they still yield greater than 20% improvements in energy consumption. The two cases for which Scale is less energy efficient than the control processor (`text` and `ospf`) are due to the extra idle power overhead that Scale consumes in these low-utilization benchmarks.

Much of the energy savings come from the control and memory access overheads that are amortized by Scale's vector-thread architecture. In comparison, other architectures—including superscalars, VLIWs, and multiprocessors—also exploit parallelism to improve performance, but only at the expense of *increased* energy consumption. Other studies have also found that vector processors are able to achieve speedups of 3–10× over high-performance superscalar processors while simultaneously using 10–50% less energy [LSCJ06].

Chapter 8

Scale Memory System Evaluation

Vector architectures provide high-throughput parallel execution resources, but much of the performance benefit they achieve comes from exposing and exploiting efficient vector memory access patterns. The vector-thread architecture allows an even broader range of applications to make use of vector loads and stores. Scale also improves on a traditional vector architecture by providing segment loads and stores to replace multiple overlapping strided accesses.

Earlier vector supercomputers [Rus78] relied on interleaved SRAM memory banks to provide high bandwidth at moderate latency, but modern vector supercomputers [D. 02, KTHK03] now implement main memory with the same commodity DRAM parts as other sectors of the computing industry, for improved cost, power, and density. As with conventional scalar processors, designers of all classes of vector machines, from vector supercomputers [D. 02] to vector microprocessors [EAE$^+$02], are motivated to add vector data caches to improve the effective bandwidth and latency of a DRAM main memory.

For vector codes with temporal locality, caches can provide a significant boost in throughput and a reduction in the energy consumed for off-chip accesses. But, even for codes with significant reuse, it is important that the cache not impede the flow of data from main memory. Existing non-blocking vector cache designs are complex since they must track all primary misses and merge secondary misses using replay queues [ASL03, Faa98]. This complexity also pervades the vector unit itself, as element storage must be reserved for all pending data accesses in either vector registers [EVS97, KP03] or a decoupled load data queue [EV96].

In this chapter, we evaluate Scale's *vector refill/access decoupling*, which is a simple and inexpensive mechanism to sustain high memory bandwidths in a cached vector machine. Scale's control processor runs ahead queuing compactly encoded commands for the vector units, and the vector refill unit (VRU) quickly pre-executes vector memory commands to detect which of the lines they will touch are not in cache and should be prefetched. This exact non-speculative hardware prefetch tries to ensure that when the vector memory instruction is eventually executed, it will find data already in cache. The cache can be simpler because we do not need to track and merge large numbers of pending misses. The vector unit is also simpler as less buffering is needed for pending element accesses. Vector refill/access decoupling is a microarchitectural technique that is invisible to software, and so can be used with any existing vector architecture.

8.1 Driving High-Bandwidth Memory Systems

The *bandwidth-delay product* for a memory system is the peak memory bandwidth, B, in bytes per processor cycle multiplied by the round-trip access latency, L, in processor cycles. To saturate a given memory system, a processor must support at least $(B/b) \times L$ independent b-byte elements in flight simultaneously. The relative latency of DRAM has been growing, while at the same time DRAM bandwidth has been rapidly improving through advances such as pipelined accesses, multiple interleaved banks, and high-speed double-data-rate interfaces. These trends combine to yield large and growing bandwidth-delay products. For example, a Pentium-4 desktop computer has around two bytes per processor cycle of main memory bandwidth with around 400 cycles of latency, representing around 200 independent 4-byte elements. A Cray X1 vector unit has around 32 bytes per cycle of global memory bandwidth with up to 800 cycles of latency across a large multiprocessor machine, representing around 3,200 independent 8-byte words [D. 02].

Each in-flight memory request has an associated hardware cost. The *access management state* is the information required to track an in-flight access, and the *reserved element data buffering* is the storage space into which the memory system will write returned data. In practice, it is impossible to stall a deeply pipelined memory system, and so element buffering must be reserved at request initiation time. The cost of supporting full memory throughput grows linearly with the bandwidth-delay product, and it is often this control overhead rather than raw memory bandwidth that limits memory system performance.

Vector machines are successful at saturating large bandwidth-delay product memory systems because they exploit *structured memory access patterns*, groups of independent memory accesses whose addresses form a simple pattern and therefore are known well in advance. Unit-stride and strided vector memory instructions compactly encode hundreds of individual element accesses in a single vector instruction [Rus78, KTHK03], and help amortize the cost of access management state across multiple elements.

At first, it may seem that a cache also exploits structured access patterns by fetching full cache lines (and thus many elements) for each request, and so a large bandwidth-delay memory system can be saturated by only tracking a relatively small number of outstanding cache line misses. Although each cache miss brings in many data elements, the processor cannot issue the request that will generate the *next* miss until it has issued all preceding requests. If these intervening requests access cache lines that are still in flight, the requests must be buffered until the line returns, and this buffering grows with the bandwidth-delay product. Any request which misses in the cache has *twice* the usual reserved element data storage: a register is reserved in the processor itself plus buffering is reserved in the cache.

Data caches amplify memory bandwidth, and in doing so they can increase the *effective* bandwidth-delay product. For example, consider a non-blocking cache with four times more bandwidth than main memory, and an application which accesses a stream of data and reuses each operand three times. The processor must now support three times the number of outstanding elements to attain peak memory throughput. With reuse, the bandwidth-delay product of a cached memory system can be as large as the cache bandwidth multiplied by the main memory latency.

8.2 Basic Access/Execute Decoupling

We first consider Scale's basic access/execute decoupling which is similar to that provided by a generic decoupled vector machine (DVM). It is useful to explain the decoupling by following a single vector load request through the system. The control processor first sends a vector load command to the VTU command queue. When it reaches the front, the command is broken into two pieces: the address portion is sent to the VLU command queue while the register writeback portion is sent to the lane command queues. The VLU then breaks the long vector load command into multiple smaller subblocks. For each subblock it reserves entries in the vector load data queues (Figure 5-19) and

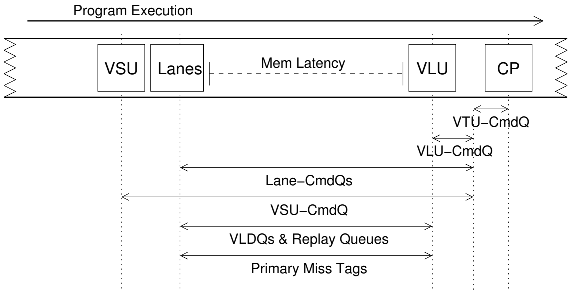


Figure 8-1: Queuing resources required in basic DVM. Decoupling allows the CP and VLU to run ahead of the lanes. Many large queues (represented by longer arrows) are needed to saturate memory.

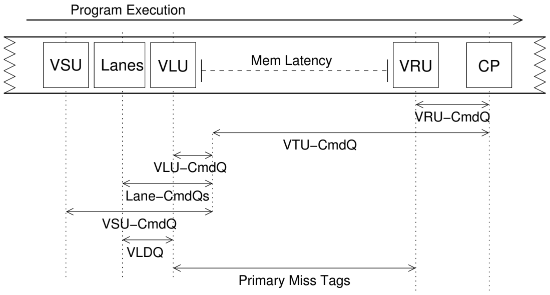


Figure 8-2: Queuing resources required in Scale. Refill/access decoupling requires fewer resources to saturate the memory system.

issues a cache request. On a hit, the data is loaded from the cache into the reserved VLDQ entries. The VLDQs enable the VLU to run ahead and issue many memory requests while the lanes trail behind and move data in FIFO order into the architecturally visible vector registers. The VLDQs also act as small memory reorder buffers since requests can return from the memory system out-of-order: hits may return while an earlier miss is still outstanding. The VSU trails behind both the VLU and the lanes and writes results to the memory

In the VTU, decoupling is enabled by the queues which buffer data and control information for the trailing units to allow the leading units to run ahead, and by the reserved element buffering provided by the VLDQs. In Scale's non-blocking cache, decoupling is enabled by the access management state in the MSHRs: *primary miss tags* and *replay queues* (described in Section 5.7.1). Figure 8-1 illustrates the resource requirements for DVM. The distance that the CP may run ahead is determined by the size of the various vector command queues. The distance that the VLU may run ahead is constrained by the lane command queues, VSU command queue, VLDQs, replay queues, and primary miss tags. The key observation is that to tolerate increasing memory latencies these resources must all be proportional to the memory bandwidth-delay product and thus can become quite large in modern memory systems.

8.3 Refill/Access Decoupling

Scale enables refill/access decoupling with its vector refill unit. The VRU receives the same commands as the VLU, and it runs ahead of the VLU issuing refill requests for the associated cache lines. The VRU implementation is described in Section 5.5.5. The primary cost is an additional address generator, and Scale's VRU implementation uses around 3000 gates, only 0.4% of the VTU total.

Ideally, the VRU runs sufficiently far ahead to allow the VLU to always hit in the cache. A refill request only brings data into the cache, and as such it needs a primary miss tag but no replay queue entry. It also does not need to provide any associated reserved element buffering in the processor, and refill requests which hit in cache due to reuse in the load stream are ignored. This evaluation assumes that stores are non-allocating and thus the VRU need not process vector store commands.

Figure 8-2 illustrates the associated resource requirements for Scale with refill/access decoupling. The VRU is able to run ahead with no need for replay queues. The VLU runs further behind the CP compared to the basic decoupled vector machine shown in Figure 8-1, and so the VTU command queue must be larger. However, since the VLU accesses are typically hits, the distance between the VLU and the lanes is shorter and the VLDQs are smaller. With refill/access decoupling, the only resources that must grow to tolerate an increasing memory latency are the VTU command queue and the number of primary miss tags. This scaling is very efficient since the VTU command queue holds vector commands which each compactly encode many element operations, and the primary miss tags must only track non-redundant cache line requests. The only reserved element buffering involved in scaling refill/access decoupling is the efficient storage provided by the cache itself.

Scale must carefully manage the relative steady-state rates of the VLU and VRU. If the VRU is able to run too far ahead it will start to evict lines which the VLU has yet to even access, and if the VLU can out-pace the VRU then the refill/access decoupling will be ineffective. In addition to the rates between the two units, Scale must also ensure that the temporal distance between them is large enough to cover the memory latency. Throttling the VLU and/or the VRU can help constrain these rates and distances to enable smoother operation. Section 5.5.5 describes Scale's throttling techniques, including the VRU's cache line *distance count* that it uses to track how far ahead of the VLU it is running. When necessary, a distance count can be tracked for each set in the cache, this is further evaluated in Section 8.6.

One disadvantage of refill/access decoupling is an increase in cache request bandwidth: the VRU first probes the cache to generate a refill and then the VLU accesses the cache to actually retrieve the data. This overhead is usually low since the VRU must only probe the cache once per cache line. Another important concern with refill/access decoupling is support for unstructured load accesses (i.e. indexed vector loads). Limited replay queuing is still necessary to provide these with some degree of access/execute decoupling.

8.4 Memory Access Benchmarks

Table 8.1 lists the benchmarks used in this evaluation. Results in the table are intended to approximately indicate each benchmark's peak performance given the processor, cache, and memory bandwidth constraints. They are based on a *pseudo-ideal* Scale configuration with very large decoupling queues, VRU refill/access decoupling, and zero cycle main memory with a peak bandwidth of 8 B/cycle. The results throughout this chapter are normalized to this performance.

The first eight benchmarks are custom kernels chosen to exercise various memory access patterns. `vvadd-word`, `vvadd-byte`, and `vvadd-cmplx` all perform vector-vector additions, with `vvadd-cmplx` making use of segment accesses to read and write the interleaved complex data. `vertex` is a graphics kernel which projects 3-D vertices in homogeneous coordinates onto a 2-D plane using four-element segment accesses to load and store the vertices. `fir` performs a 35-tap finite impulse response filter by multiplying vectors of input elements by successive tap coefficients while accumulating the output elements. Successive vector-loads are offset by one element such that each element is loaded 35 times. `transpose` uses strided segment loads and unit-stride stores to perform an out-of-place matrix transposition on word element data. The 512×512 data set size has a stride which maps columns to a single set in the cache. `idct` performs an in-place 2-D 8×8 inverse discrete cosine transform on a series of blocks using the LLM algorithm. The implementation first uses segment vector loads and stores to perform a 1-D IDCT on the block rows, then uses unit-stride accesses to perform a 1-D IDCT on the columns.

The last six benchmarks are from the EEMBC suite, in some cases with larger inputs than the reference data sets from Table 7.2. `rgbyiq` and `rgbcmyk` use three-element segment accesses to efficiently load input pixels. `rgbyiq` also uses segment accesses to store the converted pixels, but `rgbcmyk` uses a packed word store to write its output data. In `hpg`, each output pixel is a

		Access Pattern									Ops/		Elements/		Cache	Mem		
		Indexed		Unit Stride		Strided		Segment		Avg	Cycle		Cycle		Bytes/	Bytes/Cycle		
Benchmark	**Input**	**Load**	**Store**	**Load**	**Store**	**Load**	**Store**	**Load**	**Store**	**VL**	**Total**	**Muls**	**Load**	**Store**	**Cycle**	**Total**	**Load**	**Store**
`vvadd-word`	250k elements – 976.6 KB			2 W	1 W					64.0	0.67	0.00	1.33	0.67	8.00	8.00	5.33	2.67
`vvadd-byte`	1m elements – 976.6 KB			2 B	1 B					64.0	1.88	0.00	3.76	1.88	5.64	6.57	3.76	2.81
`vvadd-cmplx`	125k cmplx nums – 976.6 KB							2 W[2]:*	1 W[2]:*	32.0	0.67	0.00	1.33	0.67	7.99	8.00	5.33	2.67
`vertex`	100k 4W vertices – 1562.5 KB							1 W[4]:*	1 W[4]:*	28.0	9.55	3.82	0.96	0.96	7.64	7.65	3.82	3.82
`fir`	500k elements – 976.6 KB			1 H	1 H					124.0	7.36	3.58	3.61	0.10	7.42	0.43	0.20	0.22
`transpose400`	400×400 words – 625.0 KB				8 W			1 W[8]:1600		16.0	0.00	0.00	1.00	1.00	8.00	8.00	4.00	4.00
`transpose512`	512×512 words – 1024.0 KB				8 W			1 W[8]:2048		16.0	0.00	0.00	0.96	0.96	7.65	7.66	3.83	3.83
`idct`	20k 8×8 blocks – 2500.0 KB			8 H	8 H			1 H[8]:*	1 H[8]:*	12.0	8.80	1.45	0.96	0.96	3.86	3.85	1.93	1.92
`rgbyiq`	320×240 3B pixels – 225.0 KB							1 B[3]:*	1 B[3]:*	28.0	9.33	4.00	1.33	1.33	2.67	3.91	1.33	2.58
`rgbcmyk`	320×240 3B pixels – 225.0 KB				1 W			1 B[3]:*		32.0	6.80	0.00	1.20	0.40	2.80	3.00	1.20	1.80
`hpg`	320×240 1B pixels – 75.0 KB			3 B	1 B					63.6	10.77	3.91	2.93	0.98	3.92	2.54	1.01	1.54
`fft-ct`	4096 points – 16.0 KB		√	4 H	4 H	6 H:?	4 H:?			31.4	3.25	1.00	1.50	1.08	5.16	1.13	0.44	0.69
`rotate`	742×768 bits – 69.6 KB			8 B			8 B:-768			15.5	9.01	0.00	0.47	0.47	0.94	7.99	0.47	7.52
`dither`	256×256 1B pixels – 64.0 KB	√	√	4 H	1 H	1 B:254				31.3	5.09	0.65	1.11	0.27	2.26	0.25	0.22	0.03

Table 8.1: Refill/Access decoupling benchmark characterization. The *Access Pattern* columns display the types of memory access streams used by each benchmark. The entries are of the format $N\{B, H, W\}[n] : S$, where N indicates the number of streams of this type, B, H, or W indicates the element width (byte, half-word, or word), $[n]$ indicates the segment size in elements, and S indicates the stride in bytes between successive elements or segments. A stride of '*' indicates that the segments are consecutive, and a stride of '?' indicates that the stride changes throughout the benchmark. The *Avg VL* column displays the average vector length used by the benchmark. The per-cycle statistics are for the *pseudo-ideal* Scale configuration that is used as a baseline for normalization throughout this chapter.

function of nine input pixels, but the kernel is optimized to load each input pixel three times and hold intermediate input row computations in registers as it produces successive output rows. `fft-ct` is a radix-2 Fast Fourier Transform based on a decimation-in-time Cooley-Tukey algorithm. The algorithm inverts the inner loops after the first few butterfly stages to maximize vector lengths, resulting in a complex mix of strided and unit-stride accesses. The benchmark performs the initial bit-reversal using unit-stride loads and indexed stores. `rotate` turns a binary image by 90^o. It uses 8 unit-stride byte loads to feed an 8-bit×8-bit block rotation in registers, and then uses 8 strided byte stores to output the block. `dither` uses a strided byte access to load the input image and indexed loads and stores to read-modify-write the bit-packed output image. The dithering error buffer is updated with unit-stride accesses.

Table 8.1 shows that the kernels which operate on word data drive memory at or near the limit of 8 B/cycle. `vvadd-byte` and `fir` approach the limit of 4 load elements per cycle; and `vertex`, `fir`, `rgbyiq`, and `hpg` approach the limit of 4 multiplies per cycle. Although input data sets were chosen to be larger than the 32 KB cache, several benchmarks are able to use the cache to exploit significant temporal locality. In particular, `fir`, `dither`, and `fft-ct` have cache-to-memory bandwidth amplifications of 17×, 9×, and 4.6× respectively. Several benchmarks have higher memory bandwidth than cache bandwidth due to the non-allocating store policy and insufficient spatial-temporal locality for store writeback merging.

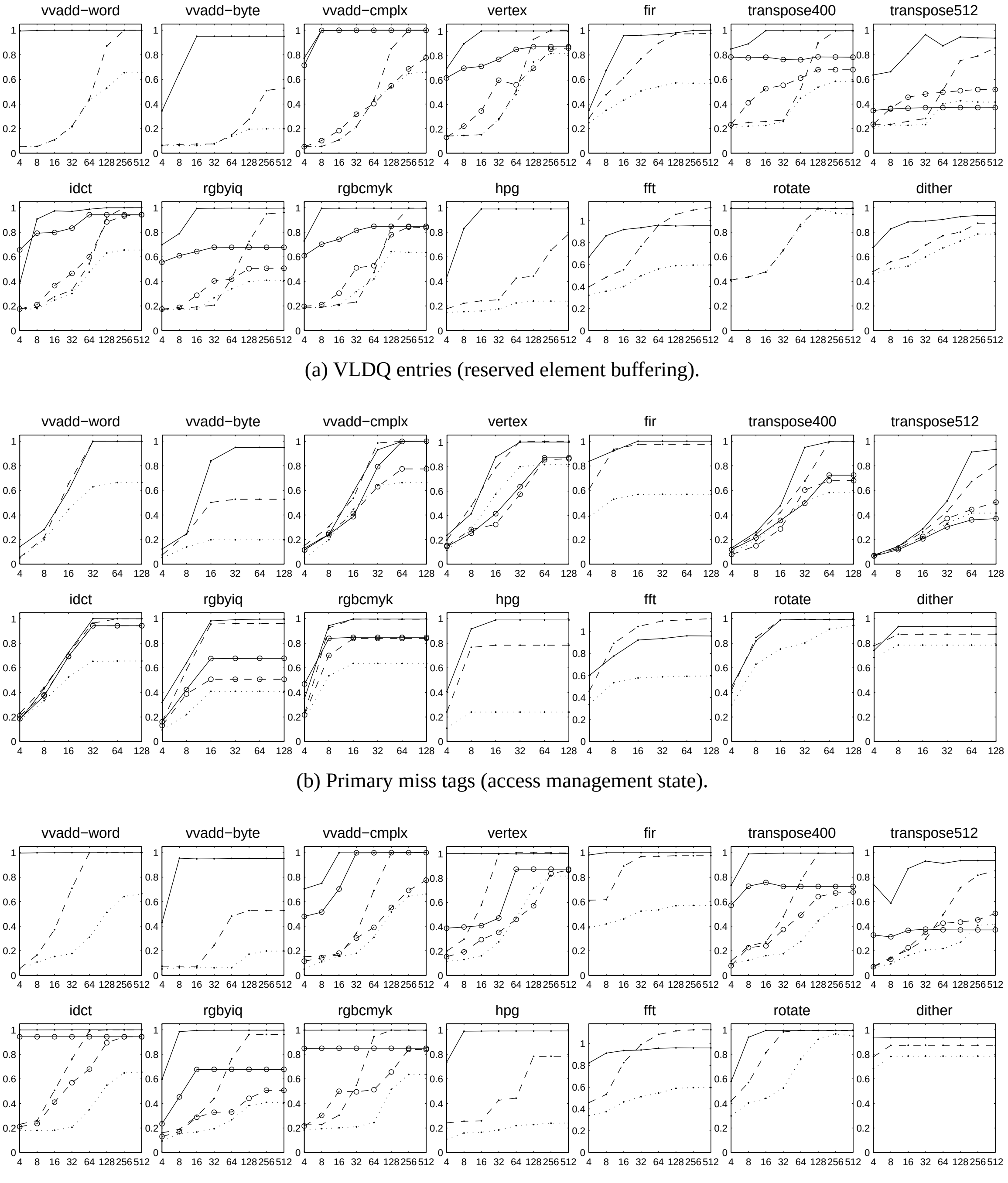
8.5 Reserved Element Buffering and Access Management State

We evaluate vector refill/access decoupling and vector segment accesses in terms of performance and resource utilization for three machine configurations: Scale with refill/access decoupling, a basic decoupled vector machine (DVM), and a decoupled scalar machine (DSM). We disable Scale's vector refill unit to model the DVM machine. For the DSM machine, we dynamically convert vector memory accesses into scalar element accesses to model an optimistic scalar processor with four load-store units and enough decoupling resources to produce hundreds of in-flight memory accesses.

Figure 8-3 presents three limit studies in which we restrict either: (a) the reserved element buffering provided by the VLDQs, (b) the number of primary miss tags, or (c) the number of replay queue entries. To isolate the requirements for each of these resources, all of the other decoupling queue sizes are set to very large values, and in each of the three studies the other two resources are unconstrained. For these limit studies, the cache uses LRU replacement, and the Scale distance throttling uses 32 distance-sets each limited to a distance of 24. Memory has a latency of 100 cycles.

The overall trend in Figure 8-3 shows that Scale is almost always able to achieve peak performance with drastically fewer reserved element buffering and access management resources compared to the DVM and DSM machines. All the machines must use primary miss tags to track many in-flight cache lines. However, in order to generate these misses, DVM and DSM also require 10s–100s of replay queue entries and reserved element buffering registers. Scale mitigates the need for these by decoupling refill requests so that demand accesses hit in the cache.

With the 100 cycle memory latency, `vvadd-word` must sustain 533 B of in-flight load data to saturate its peak load bandwidth of 5.33 B/cycle (Table 8.1). This is equivalent to about 17 cache lines, and the benchmark reaches its peak performance with 32 primary miss tags. DVM accesses four word elements with each cache access, and so requires two replay queue entries per primary miss tag (64 total) and 256 VLDQ registers to provide reserved element buffering for these in-flight accesses. In stark contrast, Scale does not require any VLDQ reserved elements or replay queue entries to generate the memory requests. DSM issues independent accesses for each of the eight words



(a) VLDQ entries (reserved element buffering).

(b) Primary miss tags (access management state).

(c) Replay queue entries (access management state).

Figure 8-3: Performance scaling with increasing reserved element buffering and access management resources. In (a), the x-axis indicates the total number of VLDQ entries across all four lanes, and in (b) and (c) the x-axis indicates the total number of primary miss tags and replay queue entries across all four cache banks. For all plots, the y-axis indicates performance relative to the *pseudo-ideal* configuration whose absolute performance is given in Table 8.1. Scale's performance is shown with solid lines, DVM with dashed lines, and DSM with dotted lines. The lines with circle markers indicate segment vector memory accesses converted into strided accesses.

in a cache line and so requires both 256 replay-queue entries and 256 reserved element buffering registers to achieve its peak performance. However, even with these resources it is still unable to saturate the memory bandwidth because, (1) each cache miss wastes a cycle of writeback bandwidth into the lanes, and (2) the lanes have bank conflicts when they issue scalar loads and stores. The other load memory bandwidth limited benchmarks share similarities with `vvadd-word` in their performance scaling.

The `vvadd-byte` kernel makes it even more difficult for DVM and DSM to drive memory since each load accesses four times less data than in `vvadd-word`. Thus, to generate a given amount of in-flight load data, they require four times more replay queue entries and VLDQ registers. Furthermore, the wasted lane writeback bandwidth and cache access bandwidth caused by misses are much more critical for `vvadd-byte`. DVM's cache misses waste half of the critical four elements per cycle lane writeback bandwidth, and so its peak performance is only half of that for Scale. To reach the peak load bandwidth, Scale uses 16 VLDQ registers (4 per lane) to cover the cache hit latency without stalling the VLU. The `rgbyiq` and `rgbcmyk` benchmarks have similar characteristics to `vvadd-byte` since they operate on streams of byte elements, but their peak throughput is limited by the processor's compute resources.

Refill/access decoupling can provide a huge benefit for access patterns with temporal reuse. The `hpg` benchmark must only sustain 1 B/cycle of load bandwidth to reach its peak throughput, and so requires 100 B of in-flight data. However, because the benchmark reads each input byte three times, DVM and DSM each require 300 reserved element buffering registers for the in-flight data. DVM accesses each 32 B cache line 24 times with four-element vector-loads, and thus requires 72 replay queue entries to sustain the three in-flight cache misses; and DSM needs four times this amount. Scale is able to use dramatically fewer resources by prefetching each cache line with a single refill request and then discarding the two subsequent redundant requests.

Refill/access decoupling can also provide a large benefit for programs that only have occasional cache misses. To achieve peak performance, `rotate`, `dither`, and `fir` must only sustain an average load bandwidth of one cache line every 68, 145, and 160 cycles respectively. However, to avoid stalling during the 100 cycles that it takes to service these occasional cache misses, DVM must use on the order of 100 VLDQ registers as reserved storage for secondary misses (`fir` and `rotate`) or to hold hit-under-miss data (`dither`). Scale attains high performance using drastically fewer resources by generating the cache misses long before the data is needed.

For benchmarks which use segment vector memory accesses, we evaluate their effect by dynamically converting each n-element segment access into n strided vector memory accesses. Figure 8-3 shows that several benchmarks rely on segment vector memory accesses to attain peak performance, even with unlimited decoupling resources. Segments can improve performance over strided accesses by reducing the number of cache accesses and bank conflicts. For example, without segments, the number of bank conflicts increases by 2.8× for `rgbcmyk`, 3.4× for `vertex`, 7.5× for `rgbyiq`, and 22× for `transpose400`. Segments also improve locality in the store stream, and thereby alleviate the need to merge independent non-allocating stores in order to optimize memory bandwidth and store buffering resources.

Although Scale always benefits from segment accesses, DVM is better off with strided accesses when reserved element buffering resources are limited. This is because a strided pattern is able to use the available VLDQ registers to generate more in-flight cache misses by first loading only one element in each segment, effectively prefetching the segment before the subsequent strided accesses return to load the neighboring elements.

Benchmark	VTU-CmdQ (size)			Total-Distance (cache lines)			Set-Distance-24 (sets)				
	1024	256	64	768	100	24	2	4	8	16	32
vvadd-word	0.03	0.03	1.00	1.00	1.00	0.32	0.63	1.00	1.00	1.00	1.00
vvadd-byte	0.02	0.95	0.90	0.96	1.00	0.48	0.92	0.94	0.98	0.98	0.92
vvadd-cmplx	0.04	0.04	0.96	1.00	1.00	0.32	0.63	1.00	1.00	1.00	1.00
vertex	0.08	0.40	0.97	0.99	1.00	0.44	0.80	1.00	1.00	1.00	0.99
idct	0.15	0.98	0.88	1.00	0.97	0.65	0.81	0.96	0.97	0.99	1.00
fir	0.99	1.00	0.99	1.00	0.97	0.79	0.84	0.92	0.99	1.00	0.99
transpose400	0.12	1.00	0.96	1.00	0.99	0.30	0.31	0.78	0.99	1.00	1.00
transpose512	0.25	0.25	0.25	0.25	0.25	0.98	1.00	1.00	1.00	1.00	1.00
rgbyiq	1.00	1.00	1.00	1.00	1.00	0.98	1.00	1.00	1.00	1.00	1.00
rgbcmyk	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
hpg	0.19	0.26	0.98	0.19	0.99	0.46	0.63	0.97	1.00	0.52	0.19
fft-ct	1.00	0.98	0.96	0.97	0.98	0.85	0.94	0.97	0.97	0.97	0.97
rotate	0.99	0.98	0.94	0.99	0.99	0.94	0.96	0.98	1.00	1.00	0.99
dither	0.98	0.96	0.96	1.00	0.95	0.61	0.71	0.87	0.93	0.99	1.00

Table 8.2: Performance for various refill distance throttling mechanisms. For each benchmark, the performance values are normalized to the best performance for that benchmark attained using any of the mechanisms.

8.6 Refill Distance Throttling

We next evaluate mechanisms for throttling the distance that the refill unit runs ahead of the VLU. Table 8.2 presents results for several throttling mechanisms at a memory latency of 400 cycles. The first group of columns evaluate having no explicit throttling other than the size of the VTU command queue (VTU-CmdQ). With a size of 1024, many of the benchmarks have dismal performance as the VRU evicts prefetched data from the cache before the VLU has a chance to access it. Simply reducing the queue size to 64 turns out to be an effective throttling technique. However, doing so can also over-constrain the refill unit, as seen with vvadd-byte and idct. In general, the command queue size is not a robust technique for throttling the refill unit since the distance that it can run ahead depends on both the vector-length and the percentage of queued commands that are vector-loads.

The *Total-Distance* and *Set-Distance-24* results in Table 8.2 evaluate directly throttling the distance that the refill unit is allowed to run ahead of the VLU. These schemes use the distance counting mechanism described in Section 5.5.5 with the *VTU-CmdQ* size set to 1024. When only one count is used to limit the VRU distance to 768 cache lines (75% of the cache) or 100 cache lines (the memory bandwidth delay product), most benchmarks perform very well. However, the refill unit still runs too far ahead for transpose512 since its accesses tend to map to a single set in the cache.

Breaking the distance count into sets allows the VRU to run further ahead when access patterns use the cache uniformly, while still not exceeding the capacity when access patterns hit a single set. The *Set-Distance-24* results in Table 8.2 use a varying granularity of distance-sets for the bookkeeping while always limiting the maximum distance in any set to 24. With a memory latency of 400 cycles, only around 4–8 distance-sets are needed.

The VRU distance throttling assumes that the cache eviction order follows its refill probe pattern, but a FIFO replacement policy can violate this assumption and cause refill/access decoupling to breakdown. This occurs when a refill probe hits a cache line which was brought into the cache long ago and is up for eviction soon. This is a problem for the hpg benchmark which has distant cache

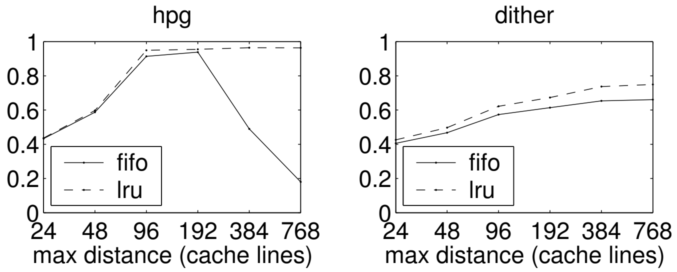


Figure 8-4: Cache replacement policy interaction with refill distance throttling. The refill unit uses the *Set-Distance-24* throttling mechanism, and the x-axis increases the maximum refill distance by varying the number of distance-sets from 1 to 32. The y-axis shows performance relative to the *pseudo-ideal* Scale configuration. The results are for a memory latency of 400 cycles.

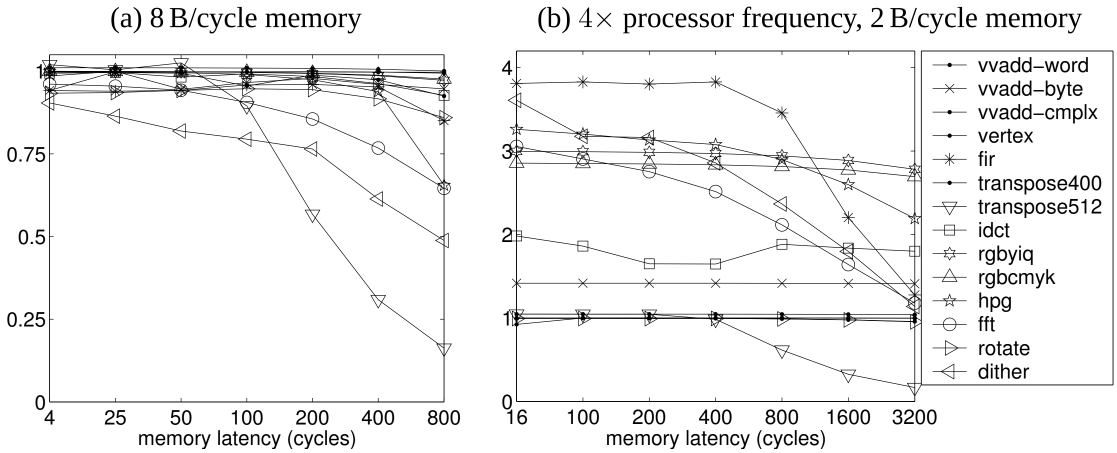


Figure 8-5: Performance scaling with increasing memory latency. The y-axis shows performance relative to the *pseudo-ideal* Scale configuration (with 8 B/cycle memory). In (b), the processor frequency is increased by 4× while retaining the same memory system.

line reuse as it processes columns of the input image. Figure 8-4 shows that its performance initially increases as the refill unit is allowed to run further ahead, but then falls off dramatically when it goes far enough to evict lines before they are reused. dither frequently reuses its pixel error buffers, but they are periodically evicted from the cache, causing the VLU to get a miss even though the corresponding refill unit probe got a hit. Figure 8-4 shows how an LRU replacement policy fixes these breakdowns in refill/access decoupling. To simplify the cache hardware, an approximation of LRU would likely be sufficient.

8.7 Memory Latency Scaling

We now evaluate a Scale configuration with reasonable decoupling queue sizes, and show how performance scales with memory latency and processor frequency. Table 8.3 shows the configuration parameters, and Figure 8-5a shows how performance scales as the latency of the 8 B/cycle main memory increases from 4 to 800 cycles. To tolerate the increasing memory bandwidth-delay product, only the VTU command queue size, the number of primary miss tags, and the distance-sets were scaled linearly with the memory latency. We found it unnecessary to scale the number of primary miss tags per bank between the 50 and 100 cycle latency configurations because applications tend to use these hard-partitioned cache bank resources more uniformly as their number increases.

Memory latency (cycles):	4	25	50	100	200	400	800
Scale Vector-Thread Unit							
Lane/VLU/VSU/VRU-CmdQs	8						
VLDQ/LDQ/SAQ	32 *(8)*						
VTU-CmdQ	4	8	16	32	64	128	256
Throttle distance per set	24						
Distance-sets	1	1	1	2	4	8	16
Non-Blocking Cache: 32 KB, 32-way set-assoc., FIFO replacement							
Replay queue entries	32 *(8)*						
Pending store buffer entries	32 *(8)*						
Primary miss tags	16*(4)*	32*(8)*	64*(16)*	64*(16)*	128*(32)*	256*(64)*	512*(128)*

Table 8.3: Resource configuration parameters with scaling memory latency. Per-lane and per-bank parameters are shown in parentheses.

As shown in Figure 8-5a, eight of the benchmarks are able to maintain essentially full throughput as the memory latency increases. This means that the amount of in-flight data reaches each benchmark's achievable memory bandwidth (Table 8.1) multiplied by 800 cycles—up to 6.25 KB. The performance for benchmarks which lack sufficient structured access parallelism (`dither` and `fft-ct`) tapers as the memory latency increases beyond their latency tolerance. `hpg` performance drops off due to the cache's FIFO eviction policy, and `transpose512` can not tolerate the long latencies since the refill distance is limited to the size of one set.

In Figure 8-5b, we evaluate the effect of increasing the processor clock frequency by $4\times$ such that the memory latency also increases by $4\times$ and the bandwidth becomes 2 B/cycle. We retain the same parameters listed in Table 8.3 for operating points with equal bandwidth-delay products. The results show that the benchmarks which were limited by memory at 8 B/cycle still attain the same performance, while those which were previously limited by compute or cache bandwidth improve with the increased frequency.

8.8 Refill/Access Decoupling Summary

We have described the considerable costs involved in supporting large numbers of in-flight memory accesses, even for vector architectures running applications that have large quantities of structured memory access parallelism. We have shown that data caches can increase performance in the presence of cache reuse, but also that this increases the effective bandwidth-delay product and hence the number of outstanding accesses required to saturate the memory system. In addition, the cost of each outstanding access increases due to the cache logic required to manage and merge pending misses. Cache refill/access decoupling dramatically reduces the cost of saturating a high-bandwidth cached memory system by pre-executing vector memory instructions to fetch data into the cache just before it is needed, avoiding most per-element costs. Furthermore, segment vector memory instructions encode common two-dimensional access patterns in a form that improves memory performance and reduces the overhead of refill/access decoupling. Together these techniques enable high-performance and low-cost vector memory systems to support the demands of modern vector architectures.

Chapter 9

Conclusion

In this thesis I have described a new architecture for an all-purpose processor core. The need for a new architecture is motivated by continuing advances in silicon technology, the emergence of high-throughput sophisticated information-processing applications, and the resulting convergence of embedded and general-purpose computing. These forces have created a growing demand for high-performance, low-power, programmable computing substrates.

I have used a three-layered approach to propose vector-thread architectures as a potential all-purpose computing solution. First, the VT architectural paradigm describes a new class of hardware/software interfaces that merge vector and multithreaded computing. This thesis compares the VT abstraction to other architectural paradigms, including vector architectures, multiprocessors, superscalars, VLIWs, and several research architectures. VT has a unique ability to expose and exploit both vector data-level parallelism and thread-level parallelism at a fine granularity, particularly when loops are mapped to the architecture. VT takes advantage of structured parallelism to achieve high performance while amortizing instruction and data overheads. The architecture is sufficiently flexible to map vectorizable loops, loops with cross-iteration dependencies, and loops with complex internal control flow. Ultimately, the vector-thread architectural paradigm is the primary contribution of this thesis, as the abstract concepts have the most potential to influence future architectures and ideas.

This thesis presents the Scale architecture as the second component of its VT proposal. As an instantiation of the vector-thread paradigm, Scale includes all of the low-level details necessary to implement an actual hardware/software interface. The Scale programmer's model defines an instruction set interface based on a virtual processor abstraction. It augments a RISC control processor with vector-configuration, vector-fetch, vector-load, and vector-store commands. Scale's virtual processors have exposed clustered execution resources where the number of private and shared registers in each cluster is configurable. VP instructions are grouped into atomic instruction blocks (AIBs), and fetch instructions allow each VP thread to direct its own control flow.

As the first implementation of VT, the Scale microarchitecture incorporates many novel mechanisms to efficiently provide vector execution and multithreading on the same hardware. Scale's vector-thread unit is organized as a set of clusters that execute independently. Cluster decoupling is enabled by a translation from VP instructions to hardware micro-ops which partition inter-cluster communication into explicit send and receive operations. The memory access cluster and the store-data cluster provide access/execute decoupling for indexed loads and stores. A lane's command management unit translates VTU commands into cluster execute directives, and it manages the small cluster AIB caches. The CMU is the meeting point for vector and threaded control, and it orchestrates the execution interleaving of the VPs on a lane. Scale's vector memory units process unit-stride and segment-strided vector-load and vector-store commands, and the vector refill unit prefetches data into the cache to provide refill/access decoupling. Execute directive chaining allows the clusters to overlap vector load data writebacks with compute operations, and the store-data cluster can also overlap vector store data reads.

The third component of the VT proposal provided by this thesis is the Scale VT processor. This hardware artifact proves that vector-thread architectures can be implemented efficiently. To build the Scale chip with limited resources, we used an automated and iterative CAD tool flow based on ASIC-style synthesis and place-and-route of standard cells. Still, we optimized area with procedural datapath pre-placement and power with extensive clock-gating. The Scale VT processor implementation demonstrates that the vector-thread control mechanisms can be implemented with compact and energy-efficient hardware structures. The Scale chip provides relatively high compute density with its 16 execution clusters and core area of 16.6 mm^2 in 180 nm technology. Its clock frequency of 260 MHz is competitive with other ASIC processors, and at this frequency the power consumption of the core is typically less than 1 W.

To support the VT proposal, this thesis also provides a detailed evaluation of Scale. The results demonstrate that many embedded benchmarks from different domains are able to achieve high performance. The code mappings exploit different types of parallelism, ranging from vectorizable loops to unstructured free-running threads. Often, vector-loads, vector-stores, and vector-fetches are interleaved at a fine granularity with threaded control flow. Execution statistics show that the Scale architecture exploits locality in both its control and data hierarchies. Measurements of the Scale chip demonstrate that, even while achieving large performance speedups, it can also be more energy efficient than a simple RISC processor.

Figure 9-1 shows a timeline history for the Scale project. The major phases and components included initial architecture development, the Scale virtual machine simulator, microarchitectural development and the scale-uarch simulator, RTL and chip design, and post-fabrication chip testing. Overall, it took about 5 years to go from the initial VT concept to the tapeout of the Scale chip.

9.1 Overall Analysis and Lessons Learned

This section highlights some of the main results and insight which come from a holistic appraisal of the Scale vector-thread architecture and implementation.

Interleaving Vector and Threaded Control. A primary outcome of this research work is the demonstration that vector and threaded control can be interleaved at a fine granularity. A vector-thread architecture allows software to succinctly and explicitly expose the parallelism in a data-parallel loop. Software uses vector commands when the control flow is uniform across the loop iterations, or threads when the loop iterations evaluate conditions independently. A vector-fetch serves as a low-overhead command for launching a set of threads. Since the threads terminate when they are done, the next vector command can usually be issued with no intervening synchronization operation.

The Scale microarchitecture and chip design demonstrate that there is not significant overhead for interleaving vector and threaded control mechanisms. Scale's execution clusters are independent processing engines with small AIB caches, and clusters are controlled by execute directives which direct them to execute AIBs from their cache. Clusters issue instructions in the same way regardless if an execute directive is from a vector-fetch or a thread-fetch. The command management unit in a lane handles the interleaving of vector commands and VP threads, and the logic that it uses to do this is only a few percent of the total lane area.

Cross-VP Communication. An important part of a VT architecture is the explicit encoding of cross-VP send and receive operations. These allow software to succinctly map loops with cross-

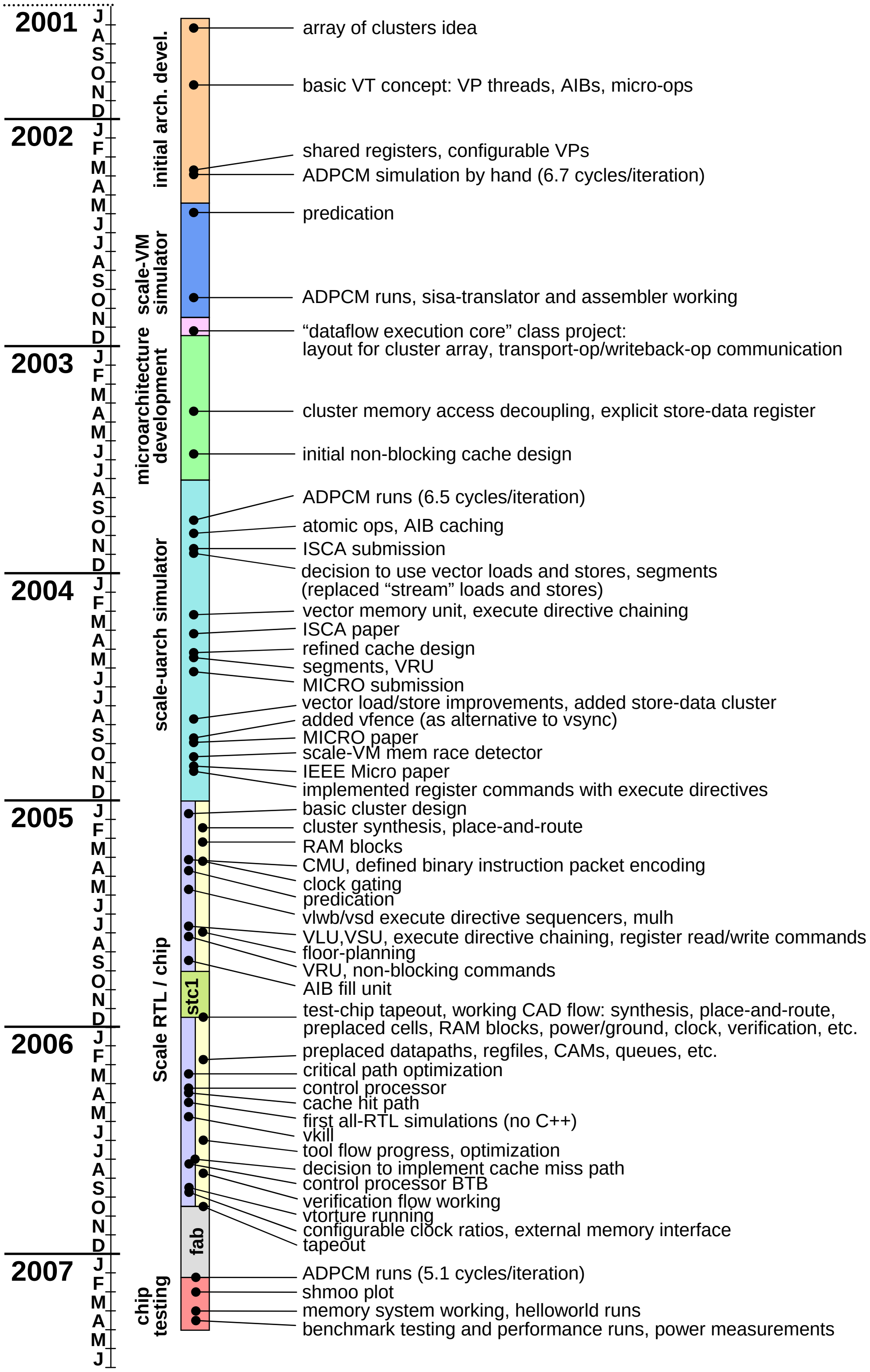

2001
J
A
S
O
N
D
2002
J
F
M
A
M
J
J
A
S
O
N
D
2003
J
F
M
A
M
J
J
A
S
O
N
D
2004
J
F
M
A
M
J
J
A
S
O
N
D
2005
J
F
M
A
M
J
J
A
S
O
N
D
2006
J
F
M
A
M
J
J
A
S
O
N
D
2007
J
F
M
A
M
J
initial arch. devel.
scale-VM simulator
microarchitecture development
scale-uarch simulator
Scale RTL / chip
stc1
fab
chip testing
array of clusters idea
basic VT concept: VP threads, AIBs, micro-ops
shared registers, configurable VPs
ADPCM simulation by hand (6.7 cycles/iteration)
predication
ADPCM runs, sisa-translator and assembler working
“dataflow execution core” class project:
layout for cluster array, transport-op/writeback-op communication
cluster memory access decoupling, explicit store-data register
initial non-blocking cache design
ADPCM runs (6.5 cycles/iteration)
atomic ops, AIB caching
ISCA submission
decision to use vector loads and stores, segments
(replaced “stream” loads and stores)
vector memory unit, execute directive chaining
ISCA paper
refined cache design
segments, VRU
MICRO submission
vector load/store improvements, added store-data cluster
added vfence (as alternative to vsync)
MICRO paper
scale-VM mem race detector
IEEE Micro paper
implemented register commands with execute directives
basic cluster design
cluster synthesis, place-and-route
RAM blocks
CMU, defined binary instruction packet encoding
clock gating
predication
vlwb/vsd execute directive sequencers, mulh
VLU,VSU, execute directive chaining, register read/write commands
floor-planning
VRU, non-blocking commands
AIB fill unit
test-chip tapeout, working CAD flow: synthesis, place-and-route,
preplaced cells, RAM blocks, power/ground, clock, verification, etc.
preplaced datapaths, regfiles, CAMs, queues, etc.
critical path optimization
control processor
cache hit path
first all-RTL simulations (no C++)
vkill
tool flow progress, optimization
decision to implement cache miss path
control processor BTB
verification flow working
vtorture running
configurable clock ratios, external memory interface
tapeout
ADPCM runs (5.1 cycles/iteration)
shmoo plot
memory system working, helloworld runs
benchmark testing and performance runs, power measurements

Figure 9-1: Scale project timeline. Various events and milestones are highlighted.

iteration dependencies and thereby expose the available parallelism between iterations. Importantly, software is not responsible for statically scheduling the parallel execution of the loop iterations. This makes the encoding compact, and it allows the hardware execution to dynamically adapt to runtime latencies. Organizing corresponding nextVP sends and prevVP receives in a vector-fetched AIB allows a single vector-fetch to establish a chain of cross-VP data transfers. This encoding also provides a means to guarantee deadlock-free execution with limited hardware queuing resources.

Atomic Instruction Blocks. AIBs are a unique component of VT architectures, but they could be a generally useful tool for building instruction set interfaces. By organizing instructions into blocks, Scale is able to amortize the overhead of explicit fetch instructions. This overhead includes the encoding space for the fetch instructions themselves, the fetch latency, and the cache tag checks and other control logic. Explicit fetches allow Scale to eliminate the need for automatic program counters, and AIBs always execute in their entirety, mitigating the need to predict branches or execute instructions speculatively. A thread can also potentially schedule its fetch instruction early in an AIB to initiate the next AIB fetch while the other instructions execute. AIBs are encoded as a unit, providing opportunities for compression, and reducing the overhead for marking the end of an AIB.

As another benefit, the atomic nature of AIB execution allows software to use temporary state that is only valid within the AIB. This improves hardware efficiency by allowing multiple threads to share the same temporary state. Scale exploits this property with its shared registers and chain registers.

Finally, AIBs are integral to the encoding of cross-VP communication, as they provide a means for grouping prevVP receives with nextVP sends.

Clusters. The Scale architecture exposes clusters to software to exploit instruction-level parallelism within a lane. Four execution clusters are provided to balance a lane's single load/store port. In mapping benchmarks to Scale, we found that it was generally not too difficult to spread local ILP across multiple clusters. In the Scale hardware, clustering of control and data paths simplifies the design and improves area and energy efficiency. The clusters each contain a small register file with few ports, reducing size and wiring energy. Static assignment of instructions to clusters allows Scale to avoid dynamic scheduling and register renaming [KP03]. Instead, each cluster executes its local compute-ops and inter-cluster transport operations in order, requiring only simple interlock logic with no dynamic inter-cluster bypass detection.

Micro-ops. Scale's decomposition of VP instructions into micro-ops proved to be a useful feature of the hardware/software interface. VP instructions provide a simple and convenient high-level software abstraction. They can target any cluster and write result data to one or more destinations in external clusters. Successive VP instructions have intuitive sequential semantics. In comparison, cluster micro-op bundles explicitly encode parallelism across clusters. They limit the number of register destination writes in a cluster to two per cycle. Explicitly partitioning inter-cluster transport and writeback operations provides the basis for independent cluster execution and decoupling. It also allows the cluster micro-op bundles in an AIB to be packed independently, in contrast to a horizontal VLIW-style encoding.

Decoupling. Vector and vector-thread architectures enable decoupling through compact commands that encode many VP (i.e. element) operations. Decoupling provides an efficient means for improving performance. Latencies can be hidden when producing operations execute sufficiently far ahead of consuming operations. Compared to centralized out-of-order processing, decoupling uses decentralized control to allow the relative execution of independent in-order units to slip dynamically.

Simple queues buffer commands and data for the trailing units, to allow the leading units to run ahead.

Decoupling is used throughout the Scale design to improve performance and efficiency. The control processor generally runs far ahead and queues compact commands for the vector-thread unit and the vector memory units. The vector refill unit pre-processes load commands to decouple cache refill from accesses by the vector-load unit. The vector load unit itself runs ahead of the lanes to fetch data from the cache, and the vector store unit runs behind the lanes and writes output data to the cache. The lane command management units and the AIB fill unit are decoupled engines which operate in the background of execution on the lanes. Compact execute directive queues allow cluster decoupling in a lane to hide functional unit, memory access, or inter-cluster transport latencies. Clusters also include decoupled transport, receive, and writeback resources to make inter-cluster scheduling more flexible. Scale partitions VP stores into address and data portions to allow cluster 0 to continue executing subsequent loads after it produces the address for a store, while the store-data cluster runs behind and waits for the data to be available.

The decoupling resources in Scale have proven to be a low-cost means for improving performance. The resulting overlap in the execution of different units efficiently exploits parallelism that is not explicit in the instruction set encoding. The decoupled structure also enables clock-gating in idle units to save power.

Multithreading. A VT architecture time-multiplexes the execution of multiple virtual processors on each physical lane. This multithreading can hide intra-thread latencies, including memory accesses and thread fetches. Scale achieves a low-cost form of simultaneous multithreading by allowing different VPs within a lane to execute at the same time in different clusters. Simultaneous multithreading is enhanced through decoupling when AIBs have acyclic inter-cluster data dependencies, and it has proven to be an important mechanism for hiding functional unit and inter-cluster transport latencies.

Structured Memory Access Patterns. To a certain extent, parallelizing computation is the simplest part of achieving high performance. The real challenge is in constructing a memory system that can supply data to keep the parallel compute units busy. Vector and vector-thread architectures provide a key advantage by exposing structured memory access patterns with vector-loads and vector-stores. Scale exploits these to amortize overheads by loading or storing multiple elements with each access. Perhaps more importantly, structured memory accesses enable decoupling to hide long latencies. Scale's vector load data queues allow the vector load unit to run ahead and fetch data from the cache while the execution clusters run behind and process the data. Scale's vector refill unit initiates cache refills early so that the VLU cache accesses are typically hits.

Scale's segment accesses have proven to be a useful feature, and software frequently uses segments to access data. Segments expose structured memory access patterns more directly by replacing interleaved strided accesses, and they improve efficiency by grouping multiple elements into each cache access. The resulting multi-element load data writebacks chain well with multi-op AIBs in the destination cluster. The Scale chip demonstrates that segment buffers fit naturally under the read and write crossbars.

Non-Blocking Multi-Banked Cache. Scale demonstrates that a relatively small 32 KB primary cache can satisfy the needs of many requesters. Sufficient bandwidth is provided by dividing the cache into four independent banks that each support wide 16-byte accesses. Additionally, the control processor and the clusters together have a total of 3.3 KB of independent instruction cache storage to optimize their local fetches. Conflicts in the primary cache are mitigated by using CAM-tags to provide 32-way set associativity.

An important requirement for a shared high-performance cache is that it continue to satisfy new accesses while misses are outstanding. A non-blocking cache is difficult to design since it must track and replay misses and handle subsequent accesses to the in-flight lines. However, a non-blocking design improves efficiency by allowing a small cache to provide high performance. Instead of relying on cache hits sustained by temporal locality (which an application may not have), the cache primarily serves as a bandwidth filter to exploit small amounts of reuse as data is continually streamed to and from memory.

Complexity of Scale. One might be struck by the complexity of the Scale architecture. Part of this impression may simply be due to the novel microarchitectural organization of a VT architecture. Other architectures, like out-of-order superscalars, are also very complicated, but their microarchitectures have become familiar to most readers. However, much of Scale's complexity comes from a fundamental trade-off between performance, efficiency, and simplicity. A design is usually a compromise between these opposing goals, and with Scale we have tried to maximize performance and efficiency. That is to say, we have tried to make Scale complexity-effective.

As an example, multi-cluster lanes add significant complexity to the Scale implementation. However, clusters improve performance by exploiting instruction-level parallelism within an lane, and exposing clusters to software improves efficiency by enabling partitioning in the hardware. Scale's translation from VP instructions to cluster micro-op bundles enables decoupling, an efficient way to boost performance.

Another source of complexity in Scale is its abstraction of physical resources as configurable virtual processors. With this interface, the hardware must manage the mapping and execution interleaving of VPs "under the hood". However, the virtualization allows Scale to improve performance and efficiency through multithreading. Also, the configurability allows Scale's hardware resources to be used more efficiently depending on the needs of the software.

Despite the complexity of the Scale design, the hardware implementation is relatively efficient in its area usage. The register files and functional units take up around 35% of the vector-thread unit area. This is a significant fraction for these baseline compute and storage resources, especially considering all of the other overheads in the VTU: the vector memory access units, the AIB caching resources (including the data RAMs, tags, control, and the AIB fill unit), the vector and thread command management and control resources (including the queues and sequencers, and the pending thread fetch queue), the transport and writeback decoupling resources, the cross-VP queues, and the miscellaneous datapath latches, muxes, and control logic. For an individual cluster, the synthesized control logic is only around 22% of the total area.

Core Size in Processor Arrays. Future all-purpose processor arrays will place a premium on compute density and efficiency. The Scale chip demonstrates that a large amount of compute, storage, and control structures can fit in a very small space. Scale's core area would shrink to only around 1 mm^2 in a modern 45 nm technology, allowing a chip to fit around 100 cores per square centimeter. In comparison, Intel's latest Penryn processor uses around 22 mm^2 per superscalar core in 45 nm technology. This order-of-magnitude difference shows that conventional general-purpose cores, which devote considerable silicon area to improving single-thread performance, are not likely to meet the needs of future high-throughput applications.

One might also consider using a scalar RISC processor as a core in a future all-purpose processor array. This is an attractive proposition since RISC cores are simple and compact. For example, the entire Scale design is around 50× larger than the control processor alone. However, a RISC processor will need instruction and data caches, say 16 KB each, and this reduces the Scale area ratio to about 5×. The connection to an inter-processor network will add even more overhead. Looking at the Scale plot in Figure 6-8, we see that the area of a cluster is about the same as the control processor. From this perspective, Scale's vector-thread unit effectively packs 16 RISC-like processing units tightly together, allowing them to amortize the area overheads of a core's cache and a network connection. Furthermore, these processing units are interconnected with a low-latency high-bandwidth network. Perhaps more important than its density of compute clusters, the Scale architecture improves efficiency compared to a multiprocessor RISC design. Scale's control processor and vector-fetch commands amortize instruction control overheads, and its vector-load and vector-store commands amortize memory access overheads. These features allow Scale to use less energy than a RISC processor when performing the same amount of work.

Related to core size is the network granularity of a processor array. Several proposals have advocated scalar operand networks to connect processing cores. With the Scale design, we argue that a core itself should provide high compute bandwidth, supported by an internal high-bandwidth low-latency scalar operand network. However, once scalar parallelism is exploited within a core, an inter-core network should optimize the bandwidth for larger data transfers. This corresponds to a software organization in which streams of data flow between coarse-grained compute kernels that are mapped to cores. The Scale results demonstrate that 32-byte cache lines are appropriate and that structured parallelism can be exploited to hide long latencies.

9.2 Future Work

There are many possible future directions for VT architectures, and the Scale design and evaluation could be extended along many dimensions. The following suggestions are just a few examples.

More Applications. The Scale evaluation would benefit from having more large applications mapped to the architecture. The 802.11a transmitter is vectorizable, while the JPEG encoder mapping has limited parallelism due to the serial block compression algorithm. It would be interesting to gain a better understanding of the fine-grained loop-level parallelism available in different types of code. Interesting applications could include video processing, graphics rendering, machine learning, and bioinformatics.

Non-Loop-Level Parallelism. Most of the Scale benchmarks mappings use either loop-level parallelization or free-running threads. Scale VP's can potentially be used to exploit other types of parallelism. One approach could be to use VPs as helper threads which process small chunks of work that a control thread offloads from a largely serial execution. Another approach could be to connect VPs in a pipeline where each performs an independent processing task.

Compilation. We designed Scale to be a compiler-friendly target, and an obvious research direction is to evaluate the effectiveness of a compiler at exploiting vector-thread parallelism. A Scale compiler is in development, and it currently has support for vectorizable loops, data-parallel loops with internal control flow, and loops with cross-iteration dependencies. The compiler uses VP thread-fetches to map conditionals and inner loops, and it uses vector-loads and vector-stores even when the body of a loop is not vectorizable. Preliminary performance results for a few benchmarks indicate that the compiler is able to achieve 5.0–5.9 VTU compute-ops per cycle on vectorizable code, and 3.5–6.8 VTU compute-ops per cycle on loops with internal control flow.

Special Compute Operations. One way to extend the Scale architecture would be to provide different types of cluster compute operations. Subword-SIMD operations could boost performance by performing multiple narrow width operations in parallel. A useful instruction could be a dual 16-bit multiply that sends two results to different destination clusters, thereby enabling two 16-bit MACs per cycle in each lane. Performance could also be improved with fixed-point instructions that include support for scaling and rounding. Additionally, support for floating point instructions would make the architecture more versatile, and these long-latency operations could put unique stresses on the microarchitecture (though Scale already supports long latency multiply and divide units).

Various special-purpose instructions could be added to speedup particular application kernels. Also, it would be interesting to add special reconfigurable units that could perform flexible bit-level computation.

Unexposing Clusters. Exposed clusters simplify hardware, but they shift the instruction-level parallelism burden to software. The alternative would be to have hardware dynamically assign instructions to clusters. An implementation could potentially use a clustered register file organization and issue micro-ops to route operands as necessary. The cluster assignment and operand management could possibly be performed during an AIB cache refill.

Larger Cores. An interesting direction for Scale could be to evaluate larger core designs. VT architectures can scale the number of lanes to improve performance while retaining binary compatibility, and many of the application kernels could benefit from 8, 16, or 32 lanes. The main hardware impediment to increasing the number of lanes would be the complexity of the crossbar. Larger designs would likely require multiple stages and have an increased latency. Unit-stride vector loads and stores could easily scale to handle more lanes. However, the cache might not provide sufficient banking to allow indexed and segment-strided accesses to issue from every lane every cycle. Segments would help alleviate this problem since they reduce the address bandwidth for a lane.

Smaller Cores. An alternative scaling approach would be to reduce the size of the Scale core, especially in the context of a chip multiprocessor. Even a single-lane Scale implementation has the potential to provide several operations per cycle of compute bandwidth while benefiting from vector-load and vector-store commands. It could be particularly attractive to try to merge the control processor and the lane hardware. The control processor could essentially be a special VP, possibly with unique scheduling priorities and VTU command capabilities. In addition to reducing hardware, such a design would unify the control processor and VP instruction sets.

VT Processor Arrays. From the beginning, Scale was intended to be an efficient core design for a processor array. One of the most interesting avenues of future work would be to evaluate a VT processor array, from both a hardware and a software perspective. Compared to other processor arrays, a VT design may achieve higher compute density, and it may simplify the software parallelization challenge by exploiting a significant amount of fine-grained parallelism within each core.

9.3 Thesis Contributions

In conclusion, this thesis makes three main contributions. The vector-thread architectural paradigm allows software to compactly encode fine-grained loop-level parallelism and locality, and VT hardware can exploit this encoding for performance and efficiency. The Scale VT architecture is a concrete instance of the VT paradigm, with a complete programmer's model and microarchitecture. Scale flexibly exposes and exploits parallelism and locality on a diverse selection of embedded benchmarks. The Scale VT processor implementation proves that vector-thread chips can achieve high performance, low power, and area efficiency.

Bibliography

[ABHS89] M. C. August, G. M. Brost, C. C. Hsiung, and A. J. Schiffleger. Cray X-MP: The Birth of a Supercomputer. *Computer*, 22(1):45–52, 1989.

[ACM88] Arvind, D. E. Culler, and G. K. Maa. Assessing the benefits of fine-grain parallelism in dataflow programs. In *Supercomputing '88: Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pages 60–69, 1988.

[AHKB00] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 248–259. ACM Press, New York, NY, USA, 2000.

[AHKW02] K. Asanovic, M. Hampton, R. Krashinsky, and E. Witchel. Energy-exposed instruction sets. pages 79–98, 2002.

[Asa98] K. Asanovic. *Vector Microprocessors*. Ph.D. thesis, EECS Department, University of California, Berkeley, 1998.

[ASL03] D. Abts, S. Scott, and D. J. Lilja. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *IPDPS*, Apr 2003.

[BA93] F. Boeri and M. Auguin. OPSILA: A Vector and Parallel Processor. *IEEE Trans. Comput.*, 42(1):76–82, 1993.

[Bar04] M. Baron. Stretching Performance. *Microprocessor Report*, 18(4), Apr 2004.

[BGGT01] A. Bik, M. Girkar, P. Grey, and X. Tian. Efficient Exploitation of Parallelism on Pentium III and Pentium 4 Processor-Based Systems. *Intel Technology Journal*, (Q1):9, 2001.

[BKA07] C. Batten, R. Krashinsky, and K. Asanovic. Scale Control Processor Test-Chip. Technical Report MIT-CSAIL-TR-2007-003, CSAIL Technical Reports, Massachusetts Institute of Technology, Jan 2007.

[BKGA04] C. Batten, R. Krashinsky, S. Gerding, and K. 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. IEEE Computer Society, Washington, DC, USA, 2004.

[CEL$^+$03] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi. The Reconfigurable Streaming Vector Processor (RSVP). In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 141. IEEE Computer Society, Washington, DC, USA, 2003.

[Cha07] M. Chang. Foundry Future: Challenges in the 21st Century. Presented at the International Solid State Circuits Conference (ISSCC) Plenary Session, Feb 2007.

[CHJ$^+$90] R. P. Colwell, W. E. Hall, C. S. Joshi, D. B. Papworth, P. K. Rodman, and J. E. Tornes. Architecture and implementation of a VLIW supercomputer. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 910–919. IEEE Computer Society, Washington, DC, USA, 1990.

[CK02] D. Chinnery and K. Keutzer. *Closing the Gap between ASIC and Custom: Tools and Techniques for High-Performance ASIC Design*. Kluwer Academic Publishers (now Springer), 2002.

[CO03] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. In *ISCA '03: Proceedings of the 30th annual International Symposium on Computer architecture*, pages 434–446. ACM Press, New York, NY, USA, 2003.

[cra03] Cray assembly language (CAL) for Cray X1 systems reference manual. Technical Report S-2314-51, Cray, Oct 2003.

[CSY90] D.-K. Chen, H.-M. Su, and P.-C. Yew. The impact of synchronization and granularity on parallel systems. In *ISCA '90: Proceedings of the 17th annual International Symposium on Computer Architecture*, pages 239–248, 1990.

[CWT$^+$01] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: long-range prefetching of delinquent loads. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 14–25. ACM Press, New York, NY, USA, 2001.

[D. 02] D. H. Brown Associates, Inc. Cray Launches X1 for Extreme Supercomputing, Nov 2002.

[DPT03] A. Duller, G. Panesar, and D. Towner. Parallel Processing - the picoChip way! In J. F. Broenink and G. H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 125–138, Sep 2003.

[DSC$^+$07] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An Integrated Quad-Core Opteron Processor. In *Proceedings of the International Solid State Circuits Conference (ISSCC)*, Feb 2007.

[EAE$^+$02] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: a vector extension to the Alpha architecture. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 281–292. IEEE Computer Society, Washington, DC, USA, 2002.

[Eat05] W. Eatherton. Keynote Address: The Push of Network Processing to the Top of the Pyramid. In *Symposium on Architectures for Networking and Communications Systems*, Oct 2005.

[EML88] C. Eoyang, R. H. Mendez, and O. M. Lubeck. The birth of the second generation: the Hitachi S-820/80. In *Supercomputing '88: Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pages 296–303. IEEE Computer Society Press, Los Alamitos, CA, USA, 1988.

[EV96] R. Espasa and M. Valero. Decoupled vector architectures. In *HPCA-2*, Feb 1996.

[EVS97] R. Espasa, M. Valero, and J. E. Smith. Out-of-order vector architectures. In *MICRO-30*, Dec 1997.

[Faa98] G. Faanes. A CMOS vector processor with a custom streaming cache. In *Proc. Hot Chips 10*, Aug 1998.

[FAD$^+$06] B. Flachs, S. Asano, S. H. Dhong, H. P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H.-J. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D. A. Brokenshire, M. Peyravian, V. To, and E. Iwata. The microarchitecture of the synergistic processor for a Cell processor. *IEEE Journal of Solid-State Circuits (JSSCC)*, 41(1):63–70, Jan 2006.

[FMJ$^+$07] J. Friedrich, B. McCredie, N. James, B. Huott, B. Curran, E. Fluhr, G. Mittal, E. Chan, Y. Chan, D. Plass, S. Chu, H. Le, L. Clark, J. Ripley, S. Taylor, J. DiLullo, and M. Lanzerotti. Design of the POWER6 Microprocessor. In *Proceedings of the International Solid State Circuits Conference (ISSCC)*, Feb 2007.

[For94] M. P. I. Forum. MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, 1994.

[Ful05] S. Fuller. The future of ASSPs: Leveraging standards for market-focused applications. Embedded Computing Design, Mar 2005.

[FWB07] X. Fan, W.-D. Weber, and L. A. Barroso. Power Provisioning for a Warehouse-sized Computer. In *ISCA-34: Proceedings of the 34th annual International Symposium on Computer Architecture*, Jun 2007.

[GBK07] A. Gangwar, M. Balakrishnan, and A. Kumar. Impact of intercluster communication mechanisms on ILP in clustered VLIW architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 12(1):1, 2007.

[Ger05] S. Gerding. *The Extreme Benchmark Suite : Measuring High-Performance Embedded Systems*. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, Sept 2005.

[GHF$^+$06] M. Gschwind, H. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, Mar 2006.

[GRE$^+$01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *WWC-4: Proceedings of the IEEE International Workshop on Workload Characterization*, pages 3–14. IEEE Computer Society, Washington, DC, USA, 2001.

[HA06] M. Hampton and K. Asanovic. Implementing virtual memory in a vector processor with software restart markers. In *ICS*, pages 135–144, 2006.

[Hal06a] T. R. Halfhill. Ambric's New Parallel Processor. *Microprocessor Report*, 20(10), Oct 2006.

[Hal06b] T. R. Halfhill. Massively Parallel Digital Video. *Microprocessor Report*, 20(1):17–22, Jan 2006.

[Hal06c] T. R. Halfhill. MathStar Challenges FPGAs. *Microprocessor Report*, 20(7), Jul 2006.

[HD86] P. Y. T. Hsu and E. S. Davidson. Highly concurrent scalar processing. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 386–395. IEEE Computer Society Press, Los Alamitos, CA, USA, 1986.

[HUYK04] S. Habata, K. Umezawa, M. Yokokawa, and S. Kitawaki. Hardware system of the earth simulator. *Parallel Comput.*, 30(12):1287–1313, 2004.

[HW97] J. R. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. pages 24–33, Apr 1997.

[IEE95] IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*, 1995.

[IEE99] IEEE. *IEEE standard 802.11a supplement. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. High-speed Physical Layer in the 5 GHz Band*, 1999.

[Int07] Intel. Press Release: Intel Details Upcoming New Processor Generations, Mar 2007.

[Jes01] C. R. Jesshope. Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines. *Australia Computer Science Communications*, 23(4):80–88, 2001.

[KBH$^+$04a] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The Vector-Thread Architecture. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 52. IEEE Computer Society, Washington, DC, USA, 2004.

[KBH$^+$04b] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The Vector-Thread Architecture. *IEEE Micro*, 24(6):84–90, Nov-Dec 2004.

[KDK$^+$01] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing with Streams. *IEEE Micro*, 21(2):35–46, 2001.

[KDM$^+$98] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee. Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture*, pages 306–317, 1998.

[KH92] G. Kane and J. Heinrich. *MIPS RISC architecture*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[Koz02] C. Kozyrakis. *Scalable vector media-processors for embedded systems*. Ph.D. thesis, University of California at Berkeley, May 2002.

[KP03] C. Kozyrakis and D. Patterson. Overcoming the Limitations of Conventional Vector Processors. In *ISCA-30*, Jun 2003.

[KPP$^+$97] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, K. Keeton, R. Thomas, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer*, 30(9):75–78, Sep 1997.

[KPP06] M. Kistler, M. Perrone, and F. Petrini. Cell Multiprocessor Communication Network: Built for Speed. *IEEE Micro*, 26(3):10–23, 2006.

[KTHK03] K. Kitagawa, S. Tagaya, Y. Hagihara, and Y. Kanoh. A Hardware Overview of SX-6 and SX-7 Supercomputer. *NEC Research & Development Journal*, 44(1):2–7, Jan 2003.

[LaP07] M. LaPedus. Sockets scant for costly ASICs. EE Times, Mar 2007.

[Lee06] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, 2006.

[LHDH05] L. Li, B. Huang, J. Dai, and L. Harrison. Automatic multithreading and multiprocessing of C programs for IXP. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 132–141. ACM Press, New York, NY, USA, 2005.

[LPMS97] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[LSCJ06] C. Lemuet, J. Sampson, J.-F. Collard, and N. Jouppi. The potential energy efficiency of vector acceleration. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 77. ACM Press, New York, NY, USA, 2006.

[Luk01] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 40–51. ACM Press, New York, NY, USA, 2001.

[McG06] H. McGhan. Niagara 2 Opens the Floodgates. *Microprocessor Report*, 20(11), Nov 2006.

[MMG$^+$06] A. Mendelson, J. Mandelblat, S. Gochman, A. Shemer, E. Niemeyer, and A. Kumar. CMP Implementation in Systems Based on the Intel Core Duo Processor. *Intel Technology Journal*, 10(2):99–107, May 2006.

[Mor68] D. R. Morrison. PATRICIA Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[MPJ$^+$00] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Proc. ISCA 27*, pages 161–171, June 2000.

[MU83] K. Miura and K. Uchida. FACOM vector processing system: VP100/200. In *Proc. NATO Advanced Research Work on High Speed Computing*, Jun 1983.

[NHS93] C. J. Newburn, A. S. Huang, and J. P. Shen. Balancing Fine- and Medium-Grained Parallelism in Scheduling Loops for the XIMD Architecture. In *Proceedings of the IFIP WG10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 39–52. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1993.

[NVI06] NVIDIA. Technical Brief: GeForce 8800 GPU Architecture Overview, Nov 2006.

[ONH$^+$96] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11. ACM Press, New York, NY, USA, 1996.

[PM93] W. Pennebaker and J. Mitchel. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, 1993.

[Pri95] C. Price. MIPS IV Instruction Set Revision 3.2. Technical report, MIPS Technologies Inc., Sep 1995.

[PWT$^+$94] M. Philippsen, T. M. Warschko, W. F. Tichy, C. G. Herter, E. A. Heinz, and P. Lukowicz. Project Triton: Towards Improved Programmability of Parallel Computers. In D. J. Lilja and P. L. Bird, editors, *The Interaction of Compilation Technology and Computer Architecture*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1994.

[QCEV99] F. Quintana, J. Corbal, R. Espasa, and M. Valero. Adding a vector unit to a superscalar processor. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 1–10. ACM Press, New York, NY, USA, 1999.

[RDK$^+$98] S. Rixner, W. Dally, U. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and J. Owens. A Bandwidth-Efficient Architecture for Media Processing. In *MICRO-31*, Nov 1998.

[RSOK06] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis. Vector Lane Threading. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 55–64. IEEE Computer Society, Washington, DC, USA, 2006.

[RTM$^+$07] S. Rusu, S. Tam, H. Muljono, D. Ayers, and J. Chang. A Dual-Core Multi-Threaded Xeon Processor with 16MB L3 Cache. In *Proceedings of the International Solid State Circuits Conference (ISSCC)*, Feb 2007.

[Rus78] R. M. Russel. The Cray-1 Computer System. *Communications of the ACM*, 21(1):63–72, Jan 1978.

[SACD04] R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The energy efficiency of CMP vs. SMT for multimedia workloads. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 196–206. ACM Press, New York, NY, USA, 2004.

[SBV95] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA-22*, pages 414–425, June 1995.

[SFS00] J. E. Smith, G. Faanes, and R. Sugumar. Vector instruction set support for conditional operations. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 260–269. ACM Press, New York, NY, USA, 2000.

[SLB06] G. Seetharaman, A. Lakhotia, and E. P. Blasch. Unmanned Vehicles Come of Age: The DARPA Grand Challenge. *Computer*, 39(12):26–29, 2006.

[Smi82] J. E. Smith. Decoupled access/execute computer architecture. In *ISCA-9*, 1982.

[SNG$^+$06] K. Sankaralingam, R. Nagarajan, P. Gratz, R. Desikan, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, W. Yoder, R. McDonald, S. Keckler, and D. Burger. The Distributed Microarchitecture of the TRIPS Prototype Processor. In *MICRO-39*, Dec 2006.

[SNL$^+$03] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. Moore. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *ISCA-30*, June 2003.

[SSK$^+$81] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, J. P T Mueller, and J. H E Smalley. PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers*, C-30(12):934–947, Dec 1981.

[Str] Stretch. White Paper: The S6000 Family of Processors.

[SUK$^+$80] M. C. Sejnowski, E. T. Upchurch, R. N. Kapur, D. P. S. Charlu, and G. J. Lipovski. An overview of the Texas Reconfigurable Array Computer. In *AFIPS 1980 National Comp. Conf.*, pages 631–641, 1980.

[THA$^+$99] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The Superthreaded Processor Architecture. *IEEE Trans. Comput.*, 48(9):881–902, 1999.

[The] The Independent JPEG Group. http://ijg.org/.

[THGM05] J. L. Tripp, A. A. Hanson, M. Gokhale, and H. Mortveit. Partitioning Hardware and Software for Reconfigurable Supercomputing Applications: A Case Study. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 27. IEEE Computer Society, Washington, DC, USA, 2005.

[TKM$^+$02] M. Taylor, J. Kim, J. Miller, D. Wentzla, F. Ghodrat, B. Greenwald, H. Ho, m Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, 22(2), 2002.

[TKM$^+$03] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, W. Lee, A. Saraf, N. Shnidman, V. Strumpen, S. Amarasinghe, and A. Agarwal. A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network. In *Proceedings of the International Solid State Circuits Conference (ISSCC)*, Feb 2003.

[TLAA05] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar Operand Networks. *IEEE Transactions on Parallel and Distributed Systems*, 16(2):145–162, 2005.

[TTG$^+$03] A. Terechko, E. L. Thenaff, M. Garg, J. van Eijndhoven, and H. Corporaal. Inter-Cluster Communication Models for Clustered VLIW Processors. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 354. IEEE Computer Society, Washington, DC, USA, 2003.

[Tur03] M. Turpin. The Dangers of Living with an X (bugs hidden in your Verilog). In *Synopsys Users Group Meeting*, Oct 2003.

[Tzi91] Tzi-cker Chiueh. Multi-threaded vectorization. In *ISCA '91: Proceedings of the 18th annual international symposium on Computer architecture*, pages 352–361. ACM Press, New York, NY, USA, 1991.

[UIT94] T. Utsumi, M. Ikeda, and M. Takamura. Architecture of the VPP500 parallel supercomputer. In *Proc. Supercomputing*, pages 478–487, Nov 1994.

[vdTJG03] E. B. van der Tol, E. G. Jaspers, and R. H. Gelderblom. Mapping of H.264 decoding on a multiprocessor architecture. In *Image and Video Communications and Processing*, 2003.

[VJM99] S. Vajapeyam, P. J. Joseph, and T. Mitra. Dynamic vectorization: a mechanism for exploiting far-flung ILP in ordinary programs. In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 16–27. IEEE Computer Society, Washington, DC, USA, 1999.

[vN93] J. von Neumann. First Draft of a Report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15(4):27–75, 1993.

[WAK$^+$96] J. Wawrzynek, K. Asanović, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan. Spert-II: A vector microprocessor system. *IEEE Computer*, 29(3):79–86, Mar 1996.

[Wat87] T. Watanabe. Architecture and performance of NEC supercomputer SX system. *Parallel Computing*, 5:247–255, 1987.

[WS91] A. Wolfe and J. 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. ACM Press, New York, NY, USA, 1991.

[Xil05] Xilinx. Press Release #0527: Xilinx Virtex-4 FX FPGAs With PowerPC Processor Deliver Breakthrough 20x Performance Boost, Mar 2005.

[Xil07] Xilinx. Data Sheet: Virtex-5 Family Overview – LX, LXT, and SXT Platforms, Feb 2007.

[YBC$^+$06] V. Yalala, D. Brasili, D. Carlson, A. Hughes, A. Jain, T. Kiszely, K. Kodandapani, A. Varadharajan, and T. Xanthopoulos. A 16-Core RISC Microprocessor with Network Extensions. In *Proceedings of the International Solid State Circuits Conference (ISSCC)*, Feb 2006.

[Yea96] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, 1996.

[YMHB00] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 225–235. ACM Press, New York, NY, USA, 2000.

[ZB91] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 712–721. ACM Press, New York, NY, USA, 1991.

[ZFMS05] H. Zhong, K. Fan, S. Mahlke, and M. 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. IEEE Computer Society, Washington, DC, USA, 2005.

[ZLM07] H. Zhong, S. Lieberman, and S. Mahlke. Extending Multicore Architectures to Exploit Hybrid Parallelism in Single-thread Applications. In *Proc. 2007 International Symposium on High Performance Computer Architecture*, February 2007.