The WaveScalar Architecture

Steven Swanson

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2006

Program Authorized to Offer Degree: Computer Science & Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Steven Swanson

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of the Supervisory Committee:

Mark Oskin

Reading Committee:

Mark Oskin

Susan Eggers

John Wawrzynek

Date:

University of Washington

Abstract

The WaveScalar Architecture

Steven Swanson

Chair of the Supervisory Committee:
Assistant Professor Mark Oskin
Computer Science & Engineering

Silicon technology will continue to provide an exponential increase in the availability of raw transistors. Effectively translating this resource into application performance, however, is an open challenge that conventional superscalar designs will not be able to meet. We present WaveScalar as a scalable alternative to conventional designs. WaveScalar is a dataflow instruction set and execution model designed for scalable, low-complexity, high-performance processors. Unlike previous dataflow machines, WaveScalar can efficiently provide the sequential memory semantics imperative languages require. To allow programmers to easily express parallelism, WaveScalar supports pthread-style, coarse-grain multithreading and dataflow-style, fine-grain threading. In addition, it permits blending the two styles within an application or even a single function.

To execute WaveScalar programs, we have designed a scalable, tile-based processor architecture called the WaveCache. As a program executes, the WaveCache maps the program's instructions onto its array of processing elements (PEs). The instructions remain at their processing elements for many invocations, and as the working set of instructions changes, the WaveCache removes unused instructions and maps new instructions in their place. The instructions communicate directly with one-another over a scalable, hierarchical on-chip interconnect, obviating the need for long wires

and broadcast communication.

This thesis presents the WaveScalar instruction set and evaluates a simulated implementation based on current technology. For single-threaded applications, the WaveCache achieves performance on par with conventional processors, but in less area. For coarse-grain threaded applications, WaveCache performance scales with chip size over a wide range, and it outperforms a range of the multi-threaded designs. The WaveCache sustains 7-14 multiply-accumulates per cycle on fine-grain threaded versions of well-known kernels. Finally, we apply both styles of threading to an example application, equake from spec2000, and speed it up by $9\times$ compared to the serial version.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

There are many, many people who have contributed directly and indirectly to this thesis. Mark Oskin provided the trifecta of mentoring, leadership, and (during the very early days of WaveScalar) Reece's Peanut Butter Cups and soda water. He also taught me to think big in research. Susan Eggers has provided invaluable guidance and support from my very first day of graduate school. She has also helped me to become a better writer.

Working with Ken Michelson, Martha Mercaldi, Andrew Petersen, Andrew Putnam, and Andrew Schwerin has been an enormous amount of fun. The vast majority of the work in this thesis would not have been possible without their hard work and myriad contributions. They are a fantastic group people. A whole slew of other grad students have helped me out in various ways during grad school: Melissa Meyer, Kurt Partridge, Mike Swift, Vibha Sazawal, Ken Yasuhara, Luke McDowell, Don Patterson, Gerome Miklau, Robert Grimm, Kevin Rennie, and Josh Redstone to name a few. I could not have asked for better colleagues (or friends) in graduate school. I fear that I may have been thoroughly spoiled.

A huge number of other people have helped me along the way, both in graduate school and before. Foremost among them are the faculty that have advised me (both on research and otherwise) through my academic career. They include Perry Fizzano, who gave me my first taste of research, Hank Levy, Roy Want, Dan Grossman, and Gaetano Boriello. I owe all my teachers from over the years a huge debt, both for everything they have taught me and for their amazing patience.

For their insightful comments on the document at hand, I thank my reading committee members: Mark Oskin, Susan Eggers, and John Wawrzynek.

Finally, my family deserves thanks above the rest. They have been unendingly supportive. They have raised me to be a nice guy, to pursue my dreams, to take it in stride when they don't work out, and to always, always keep my sense of humor.

Thanks!

26 May 2006

DEDICATION

To Glenn E. Tyler

Oh, bring back my bonnie to me.

To Spoon & Peanut

Woof!

Chapter 1
INTRODUCTION

It is widely accepted that Moore's Law will hold for the next decade. Although more transistors will be available, simply scaling up current architectures will not convert them into commensurate increases in performance [5]. The gap between the increases in performance we have come to expect and those that larger versions of existing architectures can deliver will force engineers to search for more scalable processor architectures.

Three problems contribute to this gap: (1) the ever-increasing disparity between computation and communication performance – fast transistors but slow wires; (2) the increasing cost of circuit complexity, leading to longer design times, schedule slips, and more processor bugs; and (3) the decreasing reliability of circuit technology, caused by shrinking feature sizes and continued scaling of the underlying material characteristics. In particular, modern superscalar processor designs will not scale, because they are built atop a vast infrastructure of slow broadcast networks, associative searches, complex control logic, and centralized structures.

This thesis proposes a new instruction set architecture (ISA), called WaveScalar [59], that adopts the dataflow execution model [21] to address these challenges in two ways. First, the dataflow model dictates that instructions execute when their inputs are available. Since detecting this condition can be done locally for each instruction, the dataflow model is inherently decentralized. As a result, it is well-suited to implementation in a decentralized, scalable processor. Dataflow does not require the global coordination (i.e, the program counter) that the von Neumann model relies on.

Second, the dataflow model represents programs as dataflow graphs, allowing programmers and compilers to express parallelism explicitly. Conventional high-performance von Neumann processors go to great lengths to extract parallelism from the sequence of instructions that the program counter generates. Making parallelism explicit removes this complexity from the hardware, reducing the cost of designing a dataflow machine.

WaveScalar also addresses a long standing deficiency of dataflow systems. Previous dataflow systems could not efficiently enforce the sequential memory semantics that imperative languages, such as C, C++, and Java, require. They required purely functional dataflow languages that limited dataflow's applicability to applications programmers were willing to rewrite from scratch. A recent ISCA keynote address [7] noted that for dataflow systems to be a viable alternative to von Neumann processors, they must enforce sequentiality on memory operations without severely reducing parallelism among other instructions. WaveScalar addresses this challenge with *wave-ordered memory*, a memory ordering scheme that efficiently provides the memory ordering that imperative languages need.

WaveScalar uses wave-ordered memory to replicate the memory-ordering capabilities of a conventional multi-threaded von Neumann system, but it also supports two interfaces not available in conventional systems. First, WaveScalar's fine-grain threading interface efficiently supports threads that consist of only a handful of instructions. Second, it provides an *unordered* memory interface that allows programmers to express memory parallelism. Programmers can combine both coarse- and fine-grain threads and unordered and wave-ordered memory in the same program or even the same function. Our data show that applying diverse styles of threading and memory to a single program can expose significant parallelism in code that is otherwise difficult to parallelize.

Exposing parallelism is only the first task. The processor must then translate that parallelism into performance. We exploit WaveScalar's decentralized dataflow execution model to design the *WaveCache*, a scalable, decentralized processor architecture for executing WaveScalar programs. The WaveCache has no central processing unit. Instead it consists of a substrate of simple processing elements (PEs). The WaveCache loads instructions from memory and assigns them to PEs on demand. The instructions remain at their PEs for many, potentially millions, of invocations. As the working set of instructions changes, the WaveCache evicts unneeded instructions and loads newly activated instructions in their place.

This thesis describes and evaluates the WaveScalar ISA and WaveCache architecture. First, it describes those aspects of WaveScalar's ISA (Chapters 2-3) and the WaveCache architecture (Chapter 4) that enable the execution of single-threaded applications, including the wave-ordered memory interface.

We then extend WaveScalar and the WaveCache to support conventional pthread-style threading (Chapter 5). The changes to WaveScalar include light-weight dataflow synchronization primitives and support for multiple, independent sequences of wave-ordered memory operations.

To evaluate the WaveCache, we use a wide range of workloads and tools (Chapter 6). We begin our investigation by performing a pareto analysis of the design space (Chapter 7). The analysis provides insight into WaveScalar's scalability and shows which WaveCache designs are worth building.

Once we have tuned the design, we then evaluate the performance of a small WaveCache on several single-threaded applications. The WaveCache performs comparably to a modern out-of-order superscalar design, but requires only $\sim$70% as much silicon area. For the six Splash2 [6] parallel benchmarks we use, WaveScalar achieves nearly linear speedup.

To complete the WaveScalar instruction set, we delve into WaveScalar's dataflow underpinnings (Chapter 8), the advantages they provide, and how programs can combine them with conventional multi-threading. We describe WaveScalar's "unordered" memory interface and show how it combines with fine-grain threading to reveal substantial parallelism. Finally, we hand-code three common kernels and rewrite part of the *equake* benchmark to use a combination of fine- and coarse-grain threading styles. These techniques speed up the kernels by between 16 and 240 times and *equake* by a factor of 9 compared to the serial versions.

Chapter 9 places this work in context by discussing related work, and Chapter 10 concludes.

Chapter 2
WAVESCALAR SANS MEMORY

The dataflow execution model differs fundamentally from the von Neumann model that conventional processors use. The two models use different mechanisms for representing programs, selecting instructions for execution, supporting conditional execution, communicating values between instructions, and accessing memory.

This chapter describes the features that WaveScalar inherits from past dataflow designs. It provides context for the description of the memory interface in Chapter 3 and the multi-threading facilities in Chapter 5.

For comparison, we first describe the von Neumann model and two of its limitations, namely that it is centralized and cannot express parallelism between instructions. Then we present WaveScalar's decentralized and highly-parallel dataflow model.

2.1 The von Neumann model

Von Neumann processors represent programs as a list of instructions that reside in memory. A program counter (PC) selects instructions for execution by stepping from one memory address to the next, causing each instruction to execute in turn. Special instructions can modify the PC to implement conditional execution, function calls, and other types of control transfer.

In modern von Neumann processors, instructions communicate with one another by writing and reading values in the register file. After an instruction writes a value into the register file, all subsequent instructions can read the value.

At its heart, the von Neumann model describes execution as a linear, centralized process. A single PC guides execution, and there is always exactly one instruction that, according to the model, should execute next. The model makes control transfer easy, tightly bounds the amount of state the processor must maintain, and provides a simple set of semantics. It also makes the von Neumann model an excellent match for imperative programming languages. Finally, its overwhelming commercial popularity demonstrates that constructing processors based on the model is feasible.

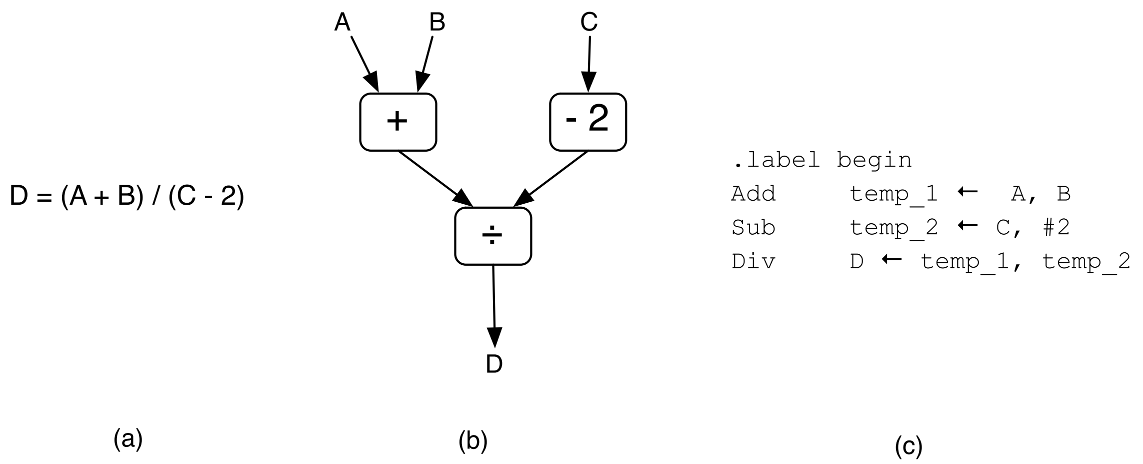
However, the von Neumann model has two key weaknesses. First, it expresses no parallelism. Second, although von Neumann processor performance has improved exponentially for over three decades, the large associative structures, deep pipelines, and broadcast-based bypassing networks they use to extract parallelism have stopped scaling [5].

2.2 WaveScalar's dataflow model

WaveScalar's dataflow model differs from the von Neumann in nearly all aspects of operation. WaveScalar represents programs as dataflow graphs instead of linear sequences of instructions. The nodes in the graph represent instructions, and communication channels or *edges* connect them. Dataflow edges carry values between instructions, replacing the register file in a von Neumann system.

Instead of a program counter, the *dataflow firing rule* [22] determines when instructions execute. The firing rule allows instructions to execute when all their inputs are available, but places no other restrictions on execution. The dataflow firing rule does not provide a total ordering on instruction execution, but does enforce the dependence-based ordering defined by the dataflow graph. Dataflow's relaxed instruction ordering allows it to exploit any parallelism in the dataflow graph, since instructions with no direct or indirect data dependences between them can execute in parallel.

Dataflow's decentralized execution model and its ability to explicitly express parallelism are its primary advantages over the von Neumann model. However, these



D = (A + B) / (C - 2)

```
.label begin
Add     temp_1 ←  A, B
Sub     temp_2 ← C, #2
Div     D ← temp_1, temp_2
```

Figure 2.1: **A simple dataflow fragment:** A simple program statement (a), its dataflow graph (b), and the corresponding WaveScalar assembly (c).

advantages do not come for free. Control transfer is more expensive in the dataflow model, and the lack of a total order on instruction execution makes it difficult to enforce the memory ordering that imperative languages require. Chapter 3 addresses this latter shortcoming.

Below we describe the aspects of WaveScalar's ISA that do not relate to the memory interface. Most of the information is not unique to WaveScalar and reflects its dataflow heritage. We present it here for completeness and to provide a thorough context for the discussion of memory ordering, which is WaveScalar's key contribution to dataflow instructions sets.

2.2.1 Program representation and execution

WaveScalar represents programs as dataflow graphs. Each node in the graph is an instruction, and the arcs between nodes encode static data dependences (i.e., dependences that are known to exist at compile time) between instructions. Figure 2.1 shows a simple piece of code, its corresponding dataflow graph, and the equivalent WaveScalar assembly language.

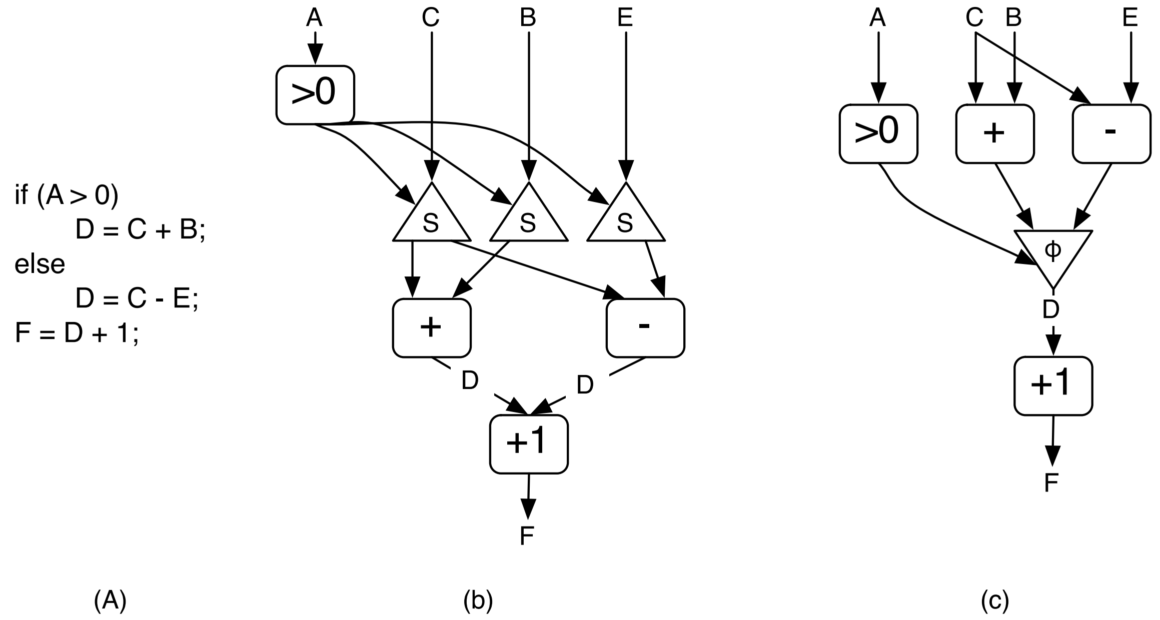
The mapping between the drawn graph and the dataflow assembly language is simple: each line of assembly represents an instruction, and the arguments to the instructions are dataflow edges. Outputs precede the '$\leftarrow$'.

The assembly code resembles RISC-style assembly but differs in two key respects. First, although the dataflow edges syntactically resemble register names, they do not correspond to a specific architectural entity. Consequently, like pseudo-registers in a compiler's program representation, there can be an arbitrary number of them. Second, the order of the instructions does not affect their execution, since they will be executed in dataflow fashion. Each instruction does have a unique address, however, used primarily for specifying function call targets (see Section 2.2.4). As in assembly languages for von Neumann machines, we can use labels (e.g., `begin` in the figure) to refer to specific instructions. We can also perform arithmetic on labels. For instance `begin` +1 refers to the SUB instruction.

Unlike the PC-driven von Neumann model, execution of a dataflow program is data-driven. Instructions execute according to the dataflow firing rule, which stipulates that an instruction can fire at any time after values arrive on all of its inputs. Instructions send the values they produce along arcs in the program's dataflow graph to their consumer instructions, causing them to fire in turn. In Figure 2.1, once inputs A and B are ready, the ADD can fire and produce the left-hand input to the DIV. Likewise, once C is available, the SUB computes the other input to the DIV instruction. The DIV then executes and produces D.

2.2.2 Control flow

Dataflow's decentralized execution algorithm makes control transfers more difficult to implement. WaveScalar provides two methods for implementing control. Instead of steering a single PC through the executable, so that the processor executes one path instead of the other, WaveScalar's STEER instruction guides values into one part of the dataflow graph and prevents them from flowing into another. Alternatively, the compiler could structure the dataflow graph to perform both computations and



```
if (A > 0)
      D = C + B;
else
      D = C - E;
F = D + 1;
```

Figure 2.2: **Implementing control in WaveScalar:** An IF-THEN-ELSE construct (a) and equivalent dataflow representations using STEER instructions (b) and ϕ instructions (c).

later discard the results on the wrong path using a ϕ instruction. In both cases, the dataflow graph must contain a control instruction for each live value, which is the source of some overhead in the form of extra static instructions.

The STEER instruction takes an input value and a boolean output selector. It directs the input to one of two possible outputs depending on the selector value, effectively steering data values to the instructions that should receive them. Figure 2.2(b) shows a simple conditional implemented with STEER instructions. STEER instructions correspond closely to traditional branch instructions. In many cases a STEER instruction can be combined with a normal arithmetic operation. For example, ADD-AND-STEER takes three inputs: a predicate and two operands, and steers the result depending on the predicate. WaveScalar provides a steering version for all 1- and 2-input instructions, eliminating 82% of the overhead STEER instructions.

The ϕ instruction [18] takes two input values and a boolean selector input and,

```
sum = 0;
for(i = 0; i < 5; i++)
   sum += i;
```

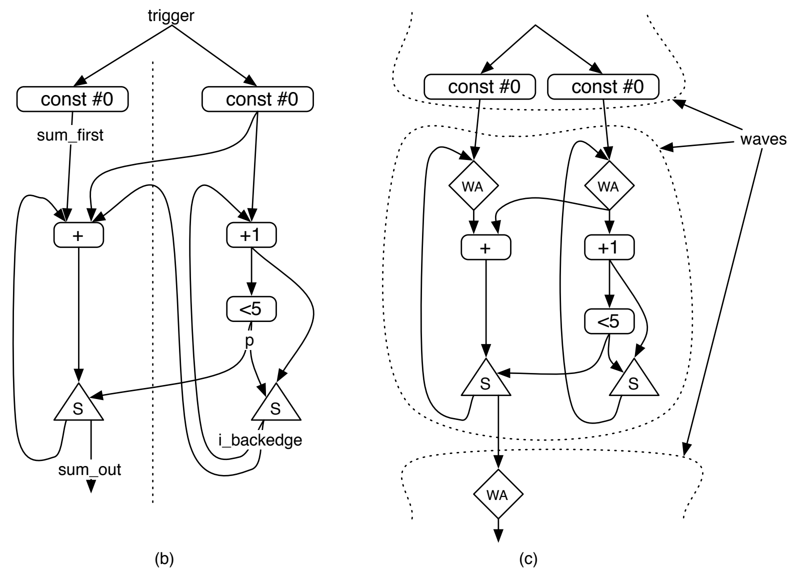


Figure 2.3: **Loops in WaveScalar:** Code for a simple loop (a), a slightly broken implementation (b), and the correct WaveScalar implementation (c).

depending on the selector, passes one of the inputs to its output. ϕ instructions are analogous to conditional moves and provide a form of predication. They remove the selector input from the critical path of some computations and therefore increase parallelism, but they waste the effort spent computing the unselected input. Figure 2.2(c) shows ϕ instructions in action.

2.2.3 Loops and waves

The Steer instruction may appear to be sufficient for WaveScalar to express loops, since it provides a basic branching facility. However, in addition to branching, dataflow machines must also distinguish dynamic instances of values from different iterations of a loop. Figure 2.3(a) shows a simple loop that illustrates the problem and WaveScalar's solution.

Execution begins when data values arrive at the Const instructions, which inject zeros into the body of the loop, one for `sum` and one for `i` (Figure 2.3(b)). On each iteration through the loop, the left side updates `sum` and the right side increments `i` and checks whether it is less than 5. For the first 5 iterations ($i = 0 \ldots 4$), `p` is true and the STEER instructions steer the new values for `sum` and `i` back into the loop. On the last iteration, `p` is false, and the final value of `sum` leaves the loop via the `sum_out` edge. Since `i` is dead after the loop, the false output of the right-side STEER instruction produces no output.

The problem arises because the dataflow execution model does not bound the latency of communication over an arc and makes no guarantee that values arrive in order. If `sum_first` takes a long time to reach the ADD instruction, the right side portion of the dataflow graph could run ahead of the left side, generating multiple values on `i_backedge` and `p`. How would the ADD and STEER instructions on the left know which of these values to use? In this particular case, the compiler could solve the problem by unrolling the loop completely, but this is not always possible.

Previous dataflow machines provided one of two solutions. In the first, *static dataflow* [21, 20], only one value is allowed on each arc at any time. In a static dataflow system, the dataflow graph as shown works fine. The processor would use back-pressure to prevent the '< 5' and '$+1$' instructions from producing a new value before the old values had been consumed. Static dataflow eliminates the ambiguity between value instances, but it reduces parallelism because multiple iterations of a loop cannot execute simultaneously. Similarly, multiple calls to a single function cannot execute in parallel, making recursion impossible.

A second model, *dynamic dataflow* [56, 30, 35, 28, 50], tags each data value with an identifier and allows multiple values to wait at the input to an instruction. The combination of a data value and its tag is called a *token*. Dynamic dataflow machines modify the dataflow firing rule so an instruction fires only when tokens *with matching tags* are available on all its inputs. WaveScalar is a dynamic dataflow architecture.

Dynamic dataflow architectures differ in how they manage and assign tags to values. In WaveScalar the tags are called *wave-numbers* [59]. We denote a WaveScalar token with wave-number W and value v as $<W>.v$. WaveScalar assigns wave numbers to compiler-defined, acyclic portions of the dataflow graph called *waves*. Waves are similar to hyperblocks [41], but they are more general, since they can contain control-flow joins and can have more than one entrance. Figure 2.3(c) shows the example loop divided into waves (as shown by the dotted lines). At the top of each wave a set of WAVE-ADVANCE instructions (the small diamonds) increments the wave numbers of the values that passes through it.

Assume the code before the loop is wave number 0. When the code executes, the two CONST instructions each produce the token <0>.0 (wave number 0, value 0). The WAVE-ADVANCE instructions take these as input and output <1>.0, so the first iteration executes in wave 1. At the end of the loop, the right-hand STEER instruction will produce <1>.1 and pass it back to the WAVE-ADVANCE at the top of its side of the loop, which will then produce <2>.1. A similar process takes place on the left side of the graph. After 5 iterations the left STEER instruction produces the final value of `sum`: <5>.10, which flows directly into the WAVE-ADVANCE at the beginning of the follow-on wave.

With the WAVE-ADVANCE instructions in place, the right side can run ahead safely, since instructions will only fire when the wave numbers in the operand tags match. More generally, waves numbers allow instructions from different wave instances, in this case iterations, to execute simultaneously.

Wave-numbers also play a key role in enforcing memory ordering (Chapter 3).

2.2.4 Function calls

Function calls on a von Neumann processor are fairly simple – the caller saves caller-saved registers, pushes function arguments and the return address onto the stack (or stores them in specific registers), and then uses a jump instruction to set the PC to the address of the beginning of the called function, triggering its execution.

Dataflow architectures adopt a different convention. WaveScalar does not need to preserve register values (there are no registers), but it must explicitly pass arguments and a return address to the function and trigger its execution.

Passing arguments to a function creates a data dependence between the caller and the callee. For indirect functions, these dependences are not statically known and therefore the dataflow graph of the application does not contain them. To compensate, WaveScalar provides an INDIRECT-SEND instruction to send a data value to an instruction at a computed address.

INDIRECT-SEND takes as input the data value to send, a base address for the destination instruction (usually a label), and the offset from that base (as an immediate). For instance, if the base address is 0x1000, and the offset is 4, INDIRECT-SEND sends the data value to the instruction at 0x1004.

Figure 2.4 contains the assembly code and dataflow graph for a small function and a call site. Dashed lines in the graph represent the dependences that exist only at run time. The LANDING-PAD instruction provides a target for a data value sent via INDIRECT-SEND[1]. The caller uses three INDIRECT-SEND instructions to call the function, two for the arguments A and B and one for the return address, which is the address of the return LANDING-PAD (label `ret` in the figure). The INDIRECT-SEND instructions compute the address of the destination instruction from the address of `foo` and their immediate values.

When the values arrive at `foo`, the LANDING-PAD instructions pass them to WAVE-ADVANCE instructions that forward them into the function body. Once the function is finished, perhaps having executed many waves, `foo` uses a single INDIRECT-SEND to return the result to the caller's LANDING-PAD instruction. After the function call, the caller starts a new wave using a WAVE-ADVANCE.

[1]LANDING-PAD instructions are not strictly necessary, since INDIRECT-SENDs can target any single-input instruction. We use them here for clarity.

```
int foo(int A, int B) {
   return A + B;
}
...
result = foo(A,B);
```

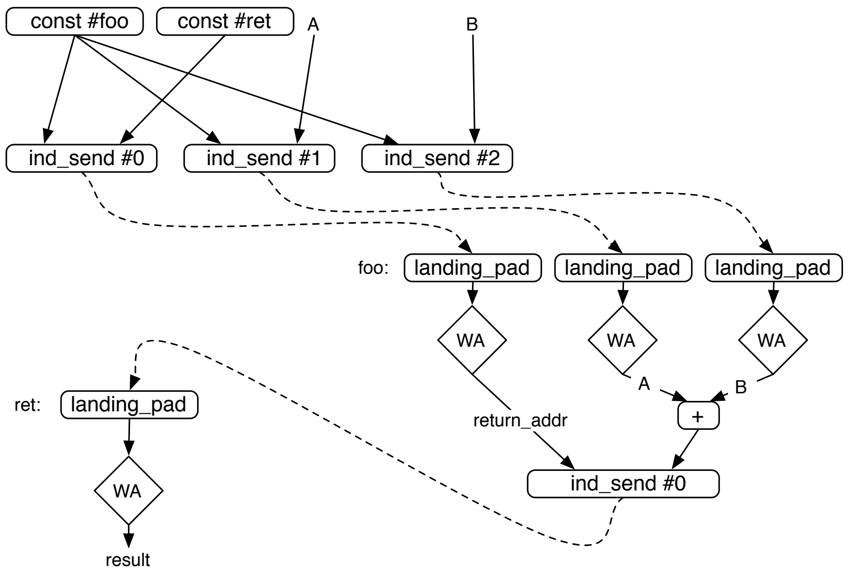
(a)

```
.label foo
Landing_Pad       return_addr_in ←
Landing_Pad       A_in ←
Landing_Pad       B_in ←
Wave_Advance      return_addr ← return_addr_in
Wave_Advance      A_addr ← A_addr_in
Wave_Advance      B ← B_in
Add               return_value ← A, B
Indirect_Send     return_value, return_addr, #0

...

Const             callee ← #foo
Const             return_addr ← #ret
Indirect_Send     return_addr, callee, #0
Indirect_Send     A, callee, #1
Indirect_Send     B, callee, #2
.label ret
Landing_Pad       result_v ←
Wave_Advance      result ← result_v
...
```

(b)



(c)

Figure 2.4: **A function call:** A simple function call (a), the assembly language version (b), and the dataflow graph (c).

2.3 Discussion

The parts of the WaveScalar's instruction set and execution model described in this chapter borrow extensively from previous dataflow systems, but they do not include facilities for accessing memory. The interface to memory is among the most important components of any architecture, because changing memory is, from the outside world's perspective, the *only* thing a processor does. The next chapter describes the fundamental incompatibility between pure dataflow execution and the linear ordering of memory operations that von Neumann processors provide and imperative languages rely upon. It also describe the interface WaveScalar provides to bridge this gap and to allow von Neumann-style programs to run efficiently on a dataflow machine.

Chapter 3
WAVE-ORDERED MEMORY

One of the largest obstacles to the success of previous dataflow architectures was their inability to efficiently execute programs written in conventional, imperative programming languages. Imperative languages require memory operations to occur in a strictly defined order to enforce data dependences through memory. Since these data dependences are implicit (i.e., they do not appear in the program's dataflow graph) the dataflow execution model does not enforce them. WaveScalar surmounts this obstacle by augmenting the dataflow model with a memory interface called *wave-ordered memory.*

Wave-ordered memory capitalizes on wave and wave-number constructs defined in Section 2.2.3. The compiler annotates memory operations within a wave with ordering information so the memory system can "chain" the operations together in the correct order. At runtime, the wave numbers provide ordering between the "chains" of operations in each wave and, therefore, across all the operations in a program.

The next section defines the ordering problem. Section 3.2 describes the wave-ordering scheme in detail. The section defines the wave-ordered memory annotations and defines an abstract memory system that provides precise semantics for wave-ordered memory. Section 3.3 describes an extension to the wave-ordered memory scheme to express parallelism between operations.

We evaluate wave-ordered memory in Section 3.4 and compare it to another proposed solution to the memory ordering problem, token-passing [12, 13]. We find that wave-ordered memory reveals twice as much memory parallelism as token-passing and investigate the reasons for the disparity.

Finally, Section 3.5 builds on the basics of wave-ordered memory to propose extensions to the system that may make it more efficient in some scenarios. It also presents sophisticated applications of wave-ordered memory aimed at maximizing memory parallelism and exploiting alias information.

The focus of this chapter is on wave-ordered memory's ability to express ordering and memory parallelism independent of the architecture that incorporates it. We do not address the hardware costs and trade-offs involved in implementing wave-ordered memory, because those issues necessarily involve the entire architecture. Chapter 5 provides a thorough discussion of these issues in the WaveScalar processor architecture and demonstrates that wave-ordered memory is practical in hardware.

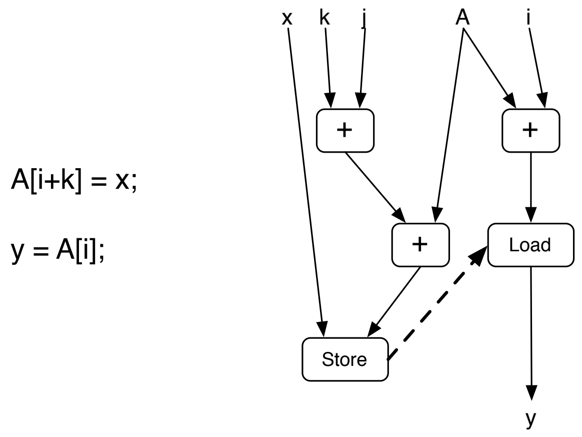
3.1 Dataflow and memory ordering

This section describes the problem of enforcing imperative language memory ordering in a dataflow environment. First, we define the ordering that imperative languages require and discuss how von Neumann machines provide it. Then we demonstrate why dataflow has trouble enforcing memory ordering.

3.1.1 Ordering in imperative languages

Imperative languages such as C and C++ provide a simple but powerful memory model. Memory is a large array of bytes that programs access, reading and writing bytes at will. To ensure correct execution, memory operations must (appear to) be applied to memory in the order the programmer specifies (i.e., in *program order*).

In a traditional von Neumann processor, the program counter provides the requisite sequencing information. It encounters memory operations as it steps through the program and feeds them into the processor's execution core. Modern processors often allow instructions to execute out-of-order, but they must guarantee that the instructions, especially memory instructions, execute in the order specified by the PC.



```
A[i+k] = x;

y = A[i];
```

Figure 3.1: **Program order:** A simple code fragment and the corresponding dataflow graph demonstrating the dataflow memory ordering problem.

3.1.2 Dataflow execution

Dataflow ISAs only enforce the static data dependences in a program's dataflow graph, because they have no mechanism to ensure that memory operations occur in program order. Figure 3.1 shows a dataflow graph that demonstrates this problem. The LOAD must execute after the STORE to ensure correct execution (dashed line), but conventional dataflow instruction sets have difficulty expressing this ordering relationship, because the relationship depend on runtime information (i.e., whether k is equal to zero).

WaveScalar is the first dataflow instruction set and execution model that can enforce the ordering between memory operations without unduely restricting the parallelism available in the dataflow graph. The next section describes wave-ordered memory and how it enforces the implicit dependences that imperative languages depend upon. Then we compare the scheme to an alternative solution and discuss potential enhancements.

3.2 Wave-ordered memory

Wave-ordered memory provides the sequential memory semantics that imperative programming languages require. Compiler-supplied annotations order the operations within each wave (Section 2.2.3) of the program. The wave numbers associated with each dynamic wave, which increase from one wave to the next, provide ordering between waves. The combination of these two orderings provides a total order on all the memory operations in the program.

Once we have described how wave-ordered memory fits into the WaveScalar model from Chapter 2, we describe the annotations the compiler provides and how they define the necessary ordering. Finally, we briefly discuss an alternative solution to the dataflow memory ordering problem.

3.2.1 Adding memory to WaveScalar

LOAD and STORE operations in WaveScalar programs fire according to the dataflow firing rule, just as normal instructions do. When they execute they send a request to the memory system that includes the current wave number and the annotations described below. In response to LOAD operations, the memory system returns the value read from memory. STORE operations do not generate a response. WaveScalar does not bound the delay between instruction execution and the request arriving at the memory system or guarantee that operations will arrive in order. Previous dataflow research refers to this type of memory interface as "split-phase" [16].

The split-phase interface provides two useful properties. First, it places few constraints on the memory system's implementation. Second, it allows us to describe the memory system without reference to the rest of the execution model. The description of wave-ordered memory below exploits this fact extensively. For the purposes of this chapter, all we need to know about WaveScalar execution is that it maintains wave numbers properly and that memory requests eventually arrive at the memory

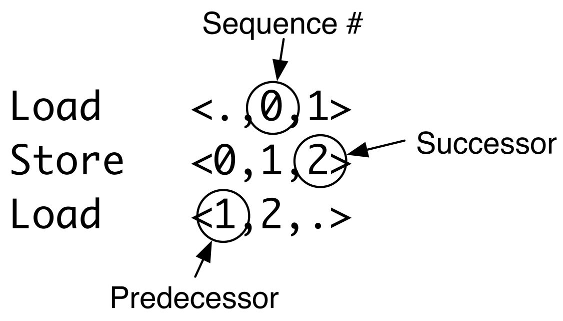


Figure 3.2: **Simple wave-ordered annotations:** The three memory operations must execute in the order shown.

interface.

3.2.2 Wave-ordering annotations

Wave-ordering annotations order the memory operations within a single wave. The annotations must guarantee two properties. First, they must ensure that the memory operations within a wave execute in the correct order. Wave-ordered memory achieves this by giving each memory operation in a wave a *sequence number*. Sequence numbers increase on all paths through a wave, ensuring that instructions with larger wave numbers come later in program order. Figure 3.2 shows a very simple series of memory operations and their annotations. The sequence number is the second of the three numbers in angle brackets. It is often convenient to refer to memory instructions by their sequence number (e.g., the first LOAD is instruction 0).

Second, wave-ordered memory must detect when all memory operations that come before a particular instruction (in program order) have executed. In the absence of branches, this is simple: since all the memory operations in a wave would eventually execute, the memory system could simply wait for memory operations with all lower sequence numbers to complete. Branches complicate this, however, because they allow instructions on the taken path to execute while those on the untaken path do not. To distinguish between operations that take a long time to fire and those that

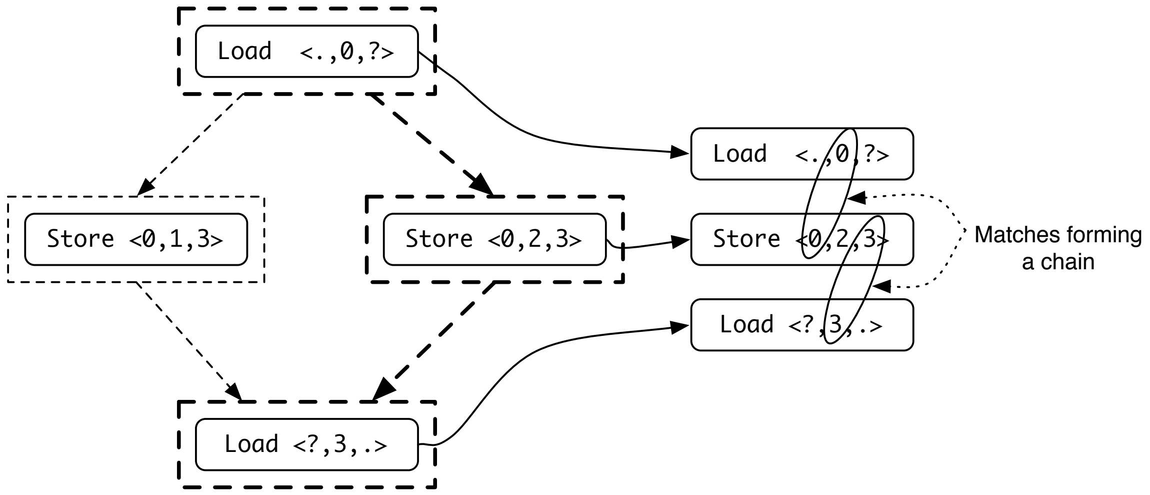


Figure 3.3: **Wave-ordering and control:** Dashed boxes and lines denote basic blocks and control paths.

never will, each memory operation also carries the sequence number of the next and previous operations in program order.

Figure 3.2 includes these annotations as well. The predecessor number is the first number between the brackets, and the successor number is the last. The Store has annotations <0, 1, 2>, because the Load with sequence number 0 precedes it and the Load with sequence number 2 follows it. The '.' symbols indicate that there is no predecessor of operation 0 and no successor of operation 2 in this wave.

At branch (join) points the successor (predecessor) number is unknown at compile time, because control may take either path. In these cases a wildcard symbol, '?,' takes the place of the successor (predecessor) number.

The left-hand portion of Figure 3.3 shows a simple if-then-else control flow graph with wildcard annotations on operations 0 and 3. The right-hand portion depicts how memory operations on the taken path are sequenced, described below.

Intuitively, the annotations allow the memory system to "chain" memory operations together. When the compiler generates and annotates a wave, there are many potential chains of operations through the wave, but only one chain (i.e., one control

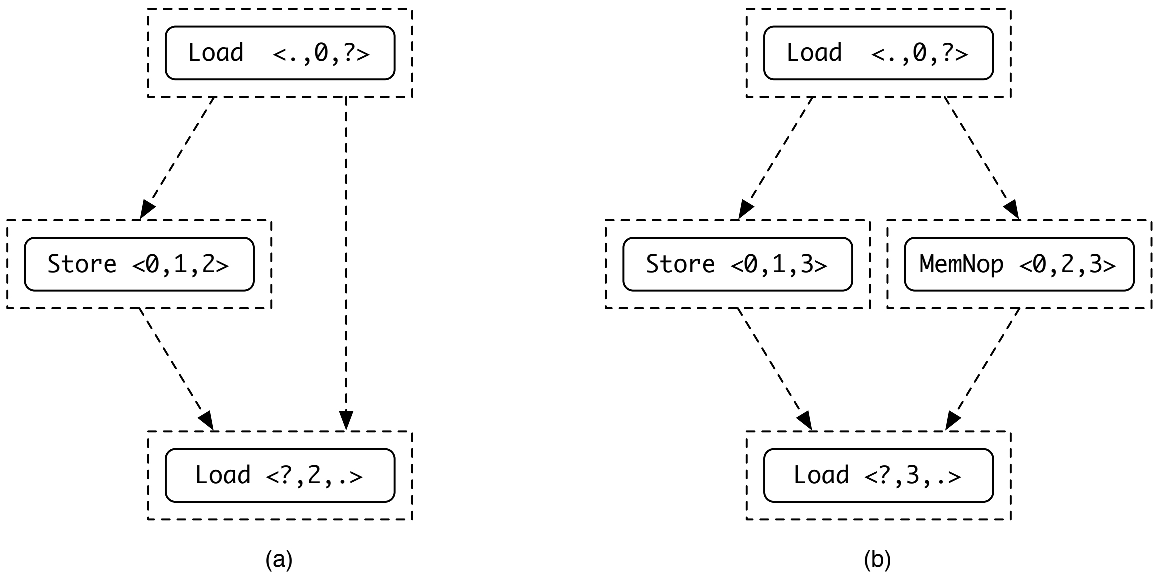


Figure 3.4: **Resolving ambiguity:** An IF-THEN without (a) and with (b) a MEMORY-NOP to allow chaining.

path) executes each time the wave executes. For instance, the right side of Figure 3.3 shows the sequence of operations along one path through the code on the left. As the ovals demonstrate, for each pair of consecutive memory instructions, either the predecessor and sequence numbers or the successor and sequence numbers match.

For the chaining to be successful, the compiler must ensure that there is a complete chain of memory operations along every path through a wave. The chain must begin with an operation whose sequence number is 0 and end with an instruction with no successor.

Ensuring that all possible chains through the wave are complete can require adding extra instructions. Figure 3.4(a) shows an example. The branch and join mean that instruction 0's successor and instruction 2's predecessor are both '?.' If control takes the right-hand path, the memory system cannot construct the required chain between operations 0 and 2. To create a chain, the compiler inserts a special MEMORY-NOP instruction between 0 and 2 on the right-hand path (Figure 3.4(b)). The MEMORY-NOP has no effect on memory but does send a request to the memory interface to provide the missing link in the chain. Adding MEMORY-NOPS introduces a small amount of overhead, usually less than 3% of static instructions.

Once the compiler annotates the operations within each wave, the wave-ordered memory system uses the information to enforce correct program behavior.

3.2.3 Formal ordering rules

To provide precise semantics for wave-ordered memory, we define an abstract memory system that defines the range of correct behavior for a program that uses wave-ordered memory. If a real memory system executes (or appears to execute) memory operations in the same order as this abstract memory system, the real memory system is, by definition, correct.

Describing an abstract memory system avoids the complications of an implementable design and does not tie the interface to a particular implementation. For example, WaveScalar's wave-ordering hardware restricts the number of sequence numbers in a wave, but other designs might not require such a constraint.

For each wave number, the abstract model maintains a list of memory requests, called an *operation buffer*. The contents of operation buffers are sorted by sequence number. During execution, requests arrive from memory instructions that have fired according to the dataflow firing rule. Memory requests may arrive in any order. When operations are ready to be applied to memory, they *issue* to the memory system and leave the operation buffer.

The *current* operation buffer is the first buffer (i.e., the buffer with the lowest wave number) that contains un-issued operations. The algorithm below guarantees that there is only one current buffer.

The predecessor, sequence, and successor numbers of a memory operation, M, are pred(M), seq(M), and succ(M), respectively. The *wavefront*, F, is the most recently issued operation in the current buffer.

When a request arrives, the wave-ordering mechanism routes it to the operation buffer for the operation's wave, inserts it into the buffer, and checks if it can issue immediately.

An operation in the current buffer can issue if one of three conditions holds:

W1 If seq(M) == 0, M may execute.

W2 If pred(M) == seq(F), M may execute.

W3 If succ(F) == seq(M), M may execute.

The first rule allows the first operation in the current wave to issue. The next two rules allow an operation to issue, if and only if, it forms a chain with the previous memory operation, F.

If the operation can issue, it does so and F is updated. The model uses the same three conditions to check if the next operation in the buffer can issue. If it can issue, it does, and the process repeats until it either reaches an operation that is not ready to issue or the end of the wave (i.e., an operation with '.' as it successor number).

If the operation cannot issue immediately, it remains in the operation buffer for later execution.

If the process reaches the of end of the current buffer, the operation buffer corresponding to the next wave becomes the current buffer, F becomes undefined, and the process continues with the first operation in the new current buffer.

In our example (Figure 3.3) each operation issues via a different condition. When the wave shown becomes the current wave, condition W1 allows operation 0 to issue. Once O has been applied to memory, condition W2 allows operation 2 to issue, since pred(2) == seq(0). Finally, operation 3 issues via condition W3, since seq(3) == succ(2).

3.2.4 Discussion

In addition to providing the semantics that imperative programming languages require, wave-ordered memory also provides the WaveScalar instruction set facility for describing the control flow graph of an application. The predecessor, successor, and sequence numbers effectively summarize the control flow graph for the memory system. To our knowledge, WaveScalar is the first instruction set to explicitly provide high-level structural information about the program. Wave-ordered memory demonstrates that hardware can exploit this type of high-level information. Providing additional information and developing architectures that can exploit it is an alluring avenue for future research.

Next, we extend wave-ordered memory so the compiler can express additional information about parallelism among memory operations.

3.3 Expressing parallelism

The annotations and rules in Section 3.2 define the strict linear ordering necessary for von Neumann-style memory semantics, but they ignore parallelism between load operations. Wave-ordered memory can express this parallelism with a fourth annotation, called a *ripple number* and denoted ripple(x), that allows consecutive Loads to issue in parallel or out-of-order.

The ripple number of a Store is equal to its sequence number. A Load's ripple number is the sequence number of the Store that most immediately precedes it. To compute the ripple number for a Load, the compiler collects the set of all Stores that precede the Load on any path through the wave. The Load's ripple number is the maximum of the Stores' sequence numbers.

Figure 3.5 shows a sequence of Load and Store operations with all four annotations. To use ripples, we add a fourth condition to the ordering rules. A Load may issue if it is next in the chain of operations (as before), *or* if the ripple number of the

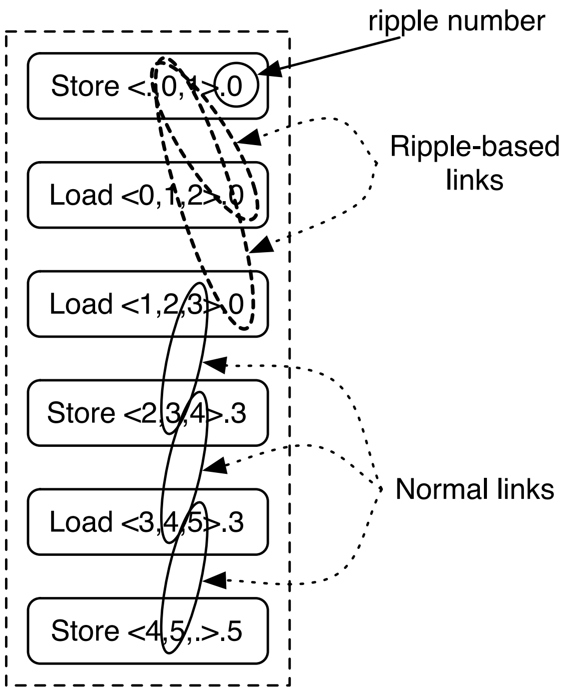


Figure 3.5: **Simple ripples:** A single wave containing a single basic block with parallelism expressed using ripple numbers.

Load is less than or equal to the sequence number of a previously issued operation (a Load or a Store). Memory-Nops are treated like Loads.

Figure 3.5 shows the two different types of links that can allow an operation to fire. The solid ovals between the bottom four operations are similar to those in Figure 3.3. The top two dashed ovals depict ripple-based links that allow the two Loads to issue in parallel.

Figure 3.6 contains a more sophisticated example. If control takes the right-side branch, Loads 1 and 4-6 can issue in parallel once Store 0 has issued, because they all have ripple numbers of 0. Load 7 must wait for one of Loads 4-6 to issue, because ripple(7) == 2 and Loads 4-6 all have sequence numbers greater than 2. If control takes the left branch, Loads 3 and 7 can issue as soon as Store 2 has issued.

Ripples also allow wave-ordered memory to exploit aliasing information that allows some stores to issue out-of-order. If we allow a Store's ripple number to differ from

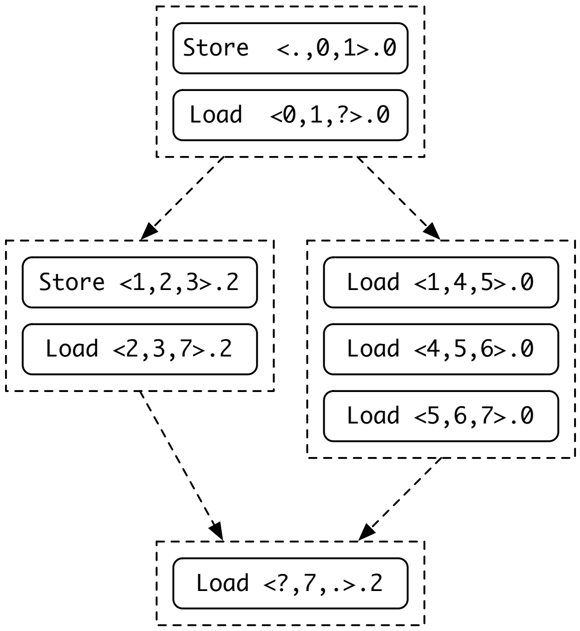


Figure 3.6: **Ripples and control:** A more sophisticated application of ripple numbers.

its sequence number, the wave-ordered interface can issue it according the same rules that apply to LOAD operations. By applying ripples to easily-analyzed STOREs (e.g., the stack or different fields in the same structure), the compiler can allow them to issue in parallel.

3.4 Evaluation

Wave-ordered memory's goal is to provide sequential memory semantics without interfering with the parallelism that dataflow exposes. Therefore, the more parallelism wave-ordered memory exposes, the better job it is doing. This section evaluates wave-ordered memory and compares it to another ordering scheme called token passing [12, 13] by measuring the amount of memory parallelism each approach reveals. It also describes an optimization that boosts wave-ordered memory's effectiveness by taking advantage of runtime information.

3.4.1 Other approaches to dataflow memory ordering

Wave-ordered memory is not the only way to sequentialize memory operations in a dataflow model. Researchers have proposed an alternative scheme that makes implicit memory dependences explicit by adding a dataflow edge between each memory operation and the next [12, 13]. While this *token passing* scheme is simple, our results show that wave-ordered memory expresses twice as much memory parallelism.

Despite this, token-passing is very useful in some situations, because it gives the programmer or compiler complete control over memory ordering. If very good memory aliasing information is available, the programmer or compiler can express parallelism directly by judiciously placing dependences only between those memory operations that must actually execute sequentially. WaveScalar provides a simple token-passing facility for just this purpose (Chapter 8).

3.4.2 Methodology

Our focus is on the expressive power of wave-ordered memory, not a particular processor architecture, so we use an idealized dataflow processor based on WaveScalar's ISA to evaluate wave-ordered memory. The processor has infinite execution resources and memory bandwidth. Memory requests travel from the instructions to the memory interface in a single cycle. Likewise data messages from one dataflow instruction to another require a single cycle.

To generate dataflow executables, we compile applications with the DEC `cc` compiler using -O4 optimizations. A binary translator-based tool-chain converts these binaries into dataflow executables. The binary translator does not perform any alias analysis and uses a simple wave-creation algorithm. The wave-ordering mechanism is a straightforward implementation of the algorithm outlined in Section 3.2. All execution is non-speculative.

To compare wave-ordered memory to token-passing, we modify the binary trans-

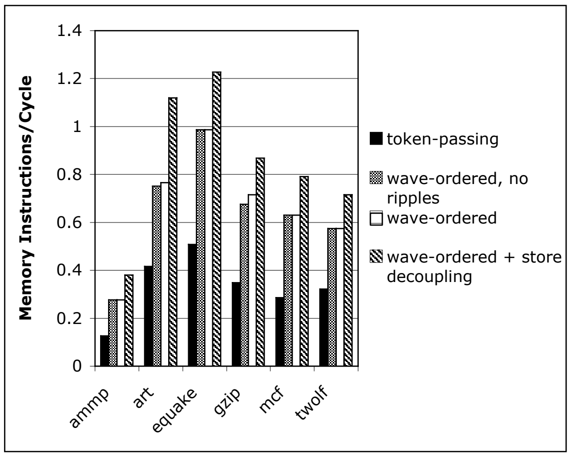


Figure 3.7: **Memory parallelism:** A comparison of Wave-ordered memory and token-passing.

lator to use a simple token-passing scheme similar to [13] instead of wave-ordered memory.

We use a selection of the Spec2000 [58] benchmark suite: *ammp*, *art*, *equake*, *gzip*, *mcf* and *twolf*. We do not use all of the spec suite due to limitations of our binary translator. We run each workload for 500 million instructions.

3.4.3 Results

Figure 3.7 shows the number of memory instructions executed per cycle (MIPC) for four different ordering schemes. The first bar in each group is token-passing. The second bar is wave-ordered memory without ripple annotations (i.e., no load parallelism) and the third bar full-blown wave-ordered memory. The next section describes the last bar.

Comparing the first two bars reveals that wave-ordered memory exposes twice as much MIPC as token-passing on average, corresponding to a 46% increase in overall IPC. The difference is primarily due to the steering dataflow machines use for control flow. The memory token is a normal data value, so the compiler must insert Steer instructions to guide it along the path of execution. The Steer instructions introduce additional latency that can delay the execution of memory operations. In our applications, there are an average of 2.5 Steer instructions between each pair of consecutive memory operations.

Wave-ordered memory avoids this overhead by encoding the necessary control flow information in the wave-ordering annotations. When the requests arrive at the memory interface they carry that information with them. The wavefront, F, from the algorithm in Section 3.2 serves the same conceptual purpose as the token: It moves from operation to operation, allowing them to fire in the correct order, but it does not traverse any non-memory instructions.

The graph also shows that between 0 and 5% of wave-ordered memory's memory parallelism comes from its ability to express load parallelism. If that is all the ripple annotations can achieve, their value is dubious. However, our binary translator performs no alias analysis and the waves it generates often contain few memory operations. We believe that a more sophisticated compiler will use ripples more effectively. Section 3.5 describes two promising approaches for generating larger waves that we will investigate in the future.

3.4.4 Decoupling store addresses and data

Wave-ordered memory can incorporate simple run-time memory disambiguation by taking advantage of the fact that the address for a Store is sometimes ready before the data value. If the memory system knew the address as soon as it was available, it could safely proceed with future memory operations to different addresses.

To incorporate this approach into wave-ordered memory, we break Store operations into two instructions, one to send the address and the other to forward the data to the memory interface.

If the store address arrives first, it is called a *partial store*. The wave-ordering mechanism treats it as a normal store operation (i.e., it issues according the rules in Section 3.2, as usual). When a partial Store issues, the memory system assigns it a *partial store queue* to hold future operations to the same address. Whenever an operation issues, the memory system checks its address against all the outstanding partial stores. If it matches one of their addresses, it goes to the end of the corresponding partial store queue. When the data for the store finally arrives, the memory system can apply all the operations in the partial store queue in quick succession.

If the data for a store arrives first, the memory system does nothing, and waits for the address to arrive. When the address arrives, the memory interface handles the store as in the normal case.

The fourth bar in Figure 3.7 shows the performance of wave-ordered memory with decoupled stores. To ensure a fair comparison, the instructions that send the store data to memory are not counted. Decoupling address and data increases memory parallelism by 30% on average and between 21% and 46% for individual applications.

3.5 Future directions

Wave-ordered memory solves the memory ordering problem for dataflow machines and outperforms token passing, but there is still room for improvement. In particular, the binary-translator constrains our current implementation of wave-ordered memory and makes implementing aggressive optimizations difficult. We are currently building a custom dataflow compiler that will be free of these constraints. This section describes several of our ideas for compiler-based improvements to wave-ordered memory. In addition, it outlines a second approach to incorporating alias analysis into wave-ordered memory.

3.5.1 Sequence number reuse

Section 3.2.2 suggests assigning each operation in a wave a unique sequence number. Since sequence numbers must be expressed in a fixed number of bits, they are a finite resource and might be scarce. The compiler can reuse sequence numbers so long as it guarantees that the sequence numbers always increase along any path and no two

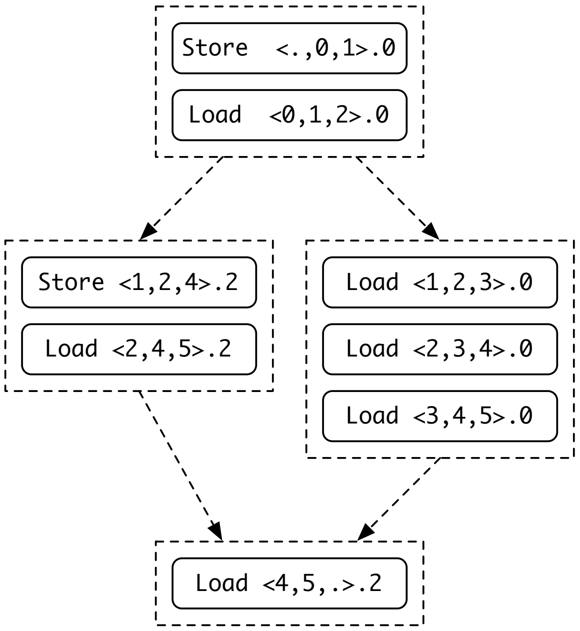


Figure 3.8: **Reusing sequence numbers:** The code in Figure 3.6 re-annotated to reduce the number of sequence numbers it requires.

operations with the same sequence number execute during one dynamic instance of the wave. Figure 3.8 shows the example in Figure 3.6 with reused sequence numbers. The figure also demonstrates a second efficiency "trick" that can eliminate '?' in some cases. Here, the sequence numbers are assigned so both sides of the branch begin (end) with an operation with sequence number 2 (4), removing the need for a wild card at both the branch and join points.

Waves with multiple entries are also possible so long as the first memory operations at each entry has sequence number 0. The next section exploits this fact to create large waves.

3.5.2 Reentrant waves

Having fewer, larger waves reduces wave-number management overhead and also enhances parallelism, because wave-ordered memory can only express load parallelism within a single wave. Two constructs interfere with producing large waves: loops and

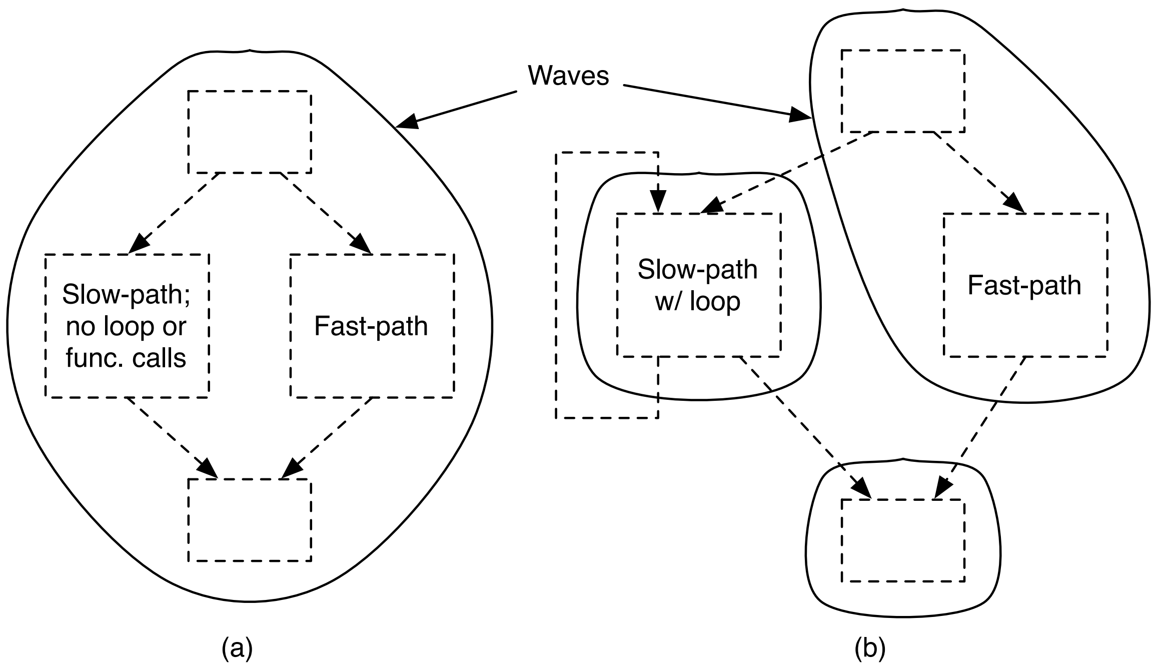


Figure 3.9: **Loops break up waves:** A block of code in a single wave (a) and broken up by a loop (b).

function calls.

Many functions contain a common case "fast-path" that executes frequently and quickly in addition to slower paths that handle exceptional or unusual cases. If the slow paths contain loops or function calls (e.g., to an error handling routine), the function can end up broken into many, small waves.

Figure 3.9 shows why this occurs. Both sides of the figure show similar code. The right-hand path is the fast-path and the left side contains infrequently executed, slow-path code. In Figure 3.9(a) the slow-path code contains ordinary instructions, so the entire code fragment is a single wave. In Figure 3.9(b) the slow-path code includes a loop. Since waves cannot contain back edges, a naive compiler might break the code into three waves, as shown. This incurs extra wave-management overhead and eliminates load-parallelism between the two waves that now make up the fast path.

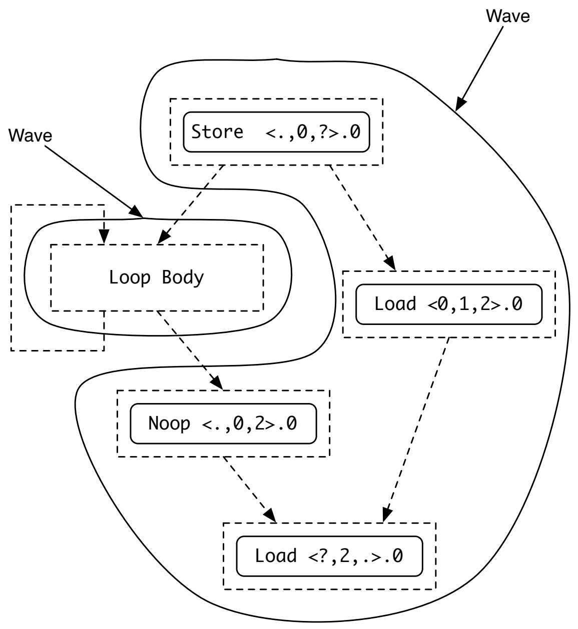


Figure 3.10: **A reentrant wave:** A wave with multiple entrances to preserve parallelism on the fast path.

Figure 3.10 demonstrates a more sophisticated solution. In the common case, control follows the right-hand path and executes in a single wave. Alternatively, control leaves the large wave, and enters the wave for the loop. When the loop is complete, control enters the large wave again but by a second entrance. Setting Memory-Nop instruction's successor number to 2 informs the memory system that operation 1 will not execute. Now, only the slow path incurs extra wave-management overhead, and the parallelism between Loads 1 and 2 remains.

3.5.3 Alias analysis

Our binary translator does not have enough information to perform aggressive alias analysis, so it is difficult to evaluate how useful ripples are for expressing the information alias analysis would provide. Other approaches to expressing parallelism may prove more effective.

One alternative approach would allow the chains of operations to "fork" and "join" so that independent sequences of operations could run in parallel. Within each parallel chain, operations would be ordered as usual. For instance, in a strongly typed language, the compiler could provide a separate chain for accessing objects of a particular type, because accesses to variables of different types are guaranteed not to conflict.

3.6 Discussion

Adding wave-ordered memory to the WaveScalar ISA in Chapter 2 provides the last piece necessary for WaveScalar to replace the von Neumann model in modern, single-threaded computer systems. The resulting instruction set is more complex than a conventional RISC ISA, but we have not found the complexity difficult to handle in our WaveScalar toolchain.

In return for the complexity, WaveScalar provides three significant benefits. First, wave-ordered memory allows WaveScalar to efficiently provide the semantics that imperative languages require and to express parallelism among LOAD operations. Second, WaveScalar can express instruction-level parallelism explicitly, while still maintaining these conventional memory semantics. Third, WaveScalar's execution model is distributed. Instructions only communicate if they must. There is no centralized control point.

In the next section we describe a microarchitecture that implements the WaveScalar ISA. We find that, in addition to increasing instruction-level parallelism, the WaveScalar instruction set allows the microarchitecture to be substantially simpler than a modern, out-of-order superscalar.

Chapter 4

A WAVESCALAR ARCHITECTURE FOR SINGLE-THREADED PROGRAMS

WaveScalar's overall goal is to enable an architecture that avoids the scaling problems described in Chapter 1. This chapter describes a tile-based WaveScalar architecture, called the *WaveCache*, that addresses those problems. The WaveCache comprises everything, except main memory, required to run a WaveScalar program. It contains a scalable grid of simple, identical dataflow processing elements, wave-ordered memory hardware, and a hierarchical interconnect to support communication. Each level of the hierarchy uses a separate communication structure: high-bandwidth, low-latency systems for local communication, and slower, narrower communication mechanisms for long distance communication.

As we will show, the WaveCache directly addresses two of the challenges we outlined in the introduction. First, the WaveCache contains no long wires. As the size of the WaveCache increases, the lengths of the longest wires do not. Second, the WaveCache architecture scales easily from small designs suitable for executing a single thread to much larger designs suited to multi-threaded workloads (see Chapter 5). The larger designs contain more tiles, but the tile structure, and therefore, the overall design complexity do not change. The challenge of defect and fault tolerance is the subject of ongoing research. The WaveCache's decentralized, uniform structure suggests that it would be easy to disable faulty components to tolerate manufacturing defects.

We begin by summarizing the WaveCache's design and operation at a high level in Section 4.1. Next, Sections 4.2 to 4.7 provide a more detailed description of its

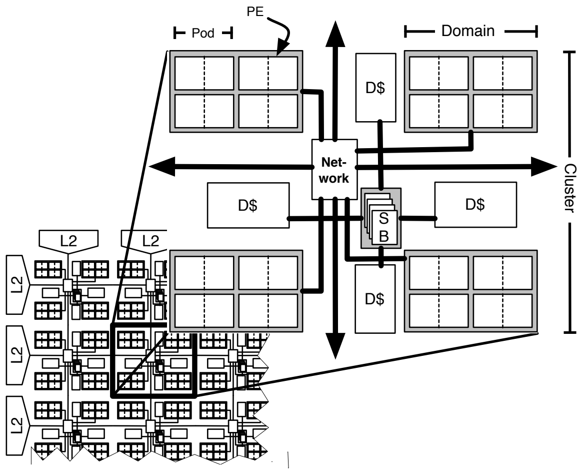


Figure 4.1: **The WaveCache:** The hierarchical organization of the microarchitecture of the WaveCache.

major components and how they interact. The next chapter evaluates the design.

4.1 WaveCache architecture overview

Several recently proposed architectures, including the WaveCache, take a tile-based approach to addressing future scaling problems [46, 53, 39, 42, 26, 13]. Instead of designing a monolithic core that comprises the entire die, tiled processors cover the die with many identical tiles, each of which is a complete, though simple, processing unit. Since they are less complex than the monolithic core and are replicated across the die, tiled architectures quickly amortize design and verification costs. Tiled architectures generally compute under decentralized control, contributing to shorter wire lengths. Finally, they can be designed to tolerate manufacturing defects in some portion of the tiles.

In the WaveCache, each tile is called a *cluster* (Figure 4.1). A cluster contains four identical *domains*, each with eight identical processing elements (PEs). In addition, each cluster has an L1 data cache, wave-ordered memory interface hardware, and a network switch for communicating with adjacent clusters.

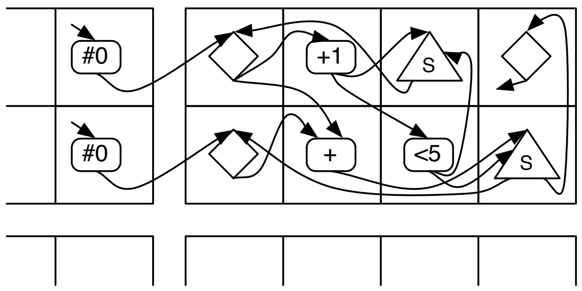


Figure 4.2: **Mapping instruction into the WaveCache:** The loop in Figure 2.3(c) mapped onto two WaveCache domains. Each large square is a processing element.

From the programmer's perspective, every static instruction in a WaveScalar binary has a dedicated processing element. Since building an array of clusters large enough to give each instruction in an entire application its own PE is impractical and wasteful, the WaveCache dynamically binds multiple instructions to a fixed number of PEs. As the working set of the application changes, the WaveCache replaces unneeded instructions with newly activated ones. In essence, the PEs cache the working set of the application, hence the WaveCache moniker.

Instructions are mapped to and placed in PEs dynamically as a program executes. The mapping algorithm has two often-conflicting goals: to minimize producer-consumer operand latency by placing dependent instructions near each other (e.g., in the same PE), and to spread independent instructions out across several PEs to exploit parallelism. Figure 4.2 illustrates how the WaveScalar program in Figure 2.3(c) can be mapped into two domains in the WaveCache. To minimize operand latency, the entire loop body resides in a single domain.

A processing element's chief responsibility is to implement the dataflow firing rule and execute instructions. Each PE contains a functional unit, specialized memories to hold operands, and logic to control instruction execution and communication. It also contains buffering and storage for several different static instructions. A PE has a five-stage pipeline, with bypass networks that allow back-to-back execution of dependent instructions at the same PE. Two aspects of the design warrant special notice. First, it avoids a large, centralized tag matching store found on some previous dataflow machines. Second, although PEs dynamically schedule execution, the scheduling hardware is dramatically simpler than a conventional dynamically scheduled processor. Section 4.2 describes the PE design in more detail.

To reduce communication costs between PEs in the processor, the architecture organizes PEs hierarchically along with their communication infrastructure (Figure 4.1). They are first coupled into *pods*; PEs within a pod snoop each others' ALU bypass networks and share instruction scheduling information, and therefore achieve the same back-to-back execution of dependent instructions as a single PE. The pods are further grouped into domains; within a domain, PEs communicate over a set of pipelined broadcast buses. The four domains in a cluster communicate point-to-point over a local switch. At the top level, clusters communicate over an on-chip interconnect built from the network switches in the clusters.

PEs access memory by sending requests to the wave-ordered memory interface in their cluster (see Section 4.4). The interface accepts the message if it is handling requests for the wave the request belongs to. Otherwise, it forwards it to the wave-ordered interface at the appropriate cluster. Section 4.4 describes the process for mapping waves to wave-ordered interfaces.

Once a memory operation issues from the wave-ordered interface, the local L1 cache provides the data, if possible. Otherwise, it initiates a conventional cache coherence request to retrieve the data from the L2 cache (located around the edge of the array of clusters, along with the coherence directory) or the L1 cache that currently owns the data.

A single cluster, combined with an L2 cache and traditional main memory, is sufficient to run any WaveScalar program, albeit possibly with poor performance as instructions are swapped in and out of the small number of available PEs. To build larger and higher performing machines, multiple clusters are connected by an on-chip network. A traditional directory-based MOESI protocol with multiple readers and a single writer maintains cache coherence.

4.2 The processing element

At a high level, the structure of a PE pipeline resembles a conventional five-stage, dynamically scheduled execution pipeline. The biggest difference is that the PE's execution is entirely data-driven. Instead of executing instructions provided by a program counter, as you find on von Neumann machines, values (i.e., tokens) arrive at a PE destined for a particular instruction. The arrival of all of an instruction's input values triggers its execution – the essence of dataflow execution.

Our goal in designing the PE was to meet our cycle-time goal while still enabling dependent instructions to execute on consecutive cycles. Pipelining was relatively simple. Back-to-back execution, however, was the source of significant complexity.

In order, the PE's pipeline stages are:

Input: Operand messages arrive at the PE either from another PE or from itself (via the ALU bypass network). The PE may reject messages if too many arrive in one cycle; the senders will then retry on a later cycle.

Match: After they leave INPUT, operands enter the *matching table*, where tag matching occurs. Cost-effective matching is essential to an efficient dataflow design and has historically been an impediment to more effective dataflow execution. The key challenge in designing the WaveCache matching table was emulating a potentially infinite table with a much smaller physical structure. This problem arises, because WaveScalar is a dynamic dataflow architecture, and places no limit on the number of dynamic instances of a static instruction that may reside in the matching table, waiting for input operands to arrive. To address this challenge, the matching table is implemented as a specialized cache for a larger in-memory matching table – a common dataflow technique [30, 56].

The matching table is associated with a *tracker board*, which determines when an instruction has a complete set of inputs, and is therefore ready to execute. When this occurs, the instruction moves into the scheduling queue.

Dispatch: The PE selects an instruction from the scheduling queue, reads its operands from the matching table and forwards them to EXECUTE. If the destination of the dispatched instruction is local, it speculatively issues the consumer instruction to the scheduling queue, enabling its execution on the next cycle.

Execute: In most cases EXECUTE executes an instruction and sends its results to OUTPUT, which broadcasts it over the bypass network. However, there are two cases in which execution will not occur. First, if an instruction was dispatched speculatively and one of its operands has not yet arrived, the instruction is squashed. Second, if OUTPUT is full, EXECUTE stalls until space becomes available.

Output: Instruction outputs are sent via the output bus to their consumer instructions, either at this PE or a remote PE. The output buffer broadcasts the value on the PE's broadcast bus. In the common case, a consumer PE within that domain accepts the value immediately. It is possible, however, that the consumer cannot handle the value that cycle and will reject it. The round-trip to send the value and receive an ACK/NACK reply takes four cycles. Rather than have the data value occupy the output register for that period, the PE assumes it will be accepted, moving it into its 4-entry *reject buffer*, and inserts a new value into the output buffer on the next cycle. If an operand ends up being rejected, it is fed back into the output queue to be sent again to the destinations that rejected it. When all the receivers have accepted the value, the reject buffer discards it.

Figure 4.3 illustrates how instructions from a simple dataflow graph (on the left side of the figure) flow through the WaveCache pipeline. It also illustrates how the

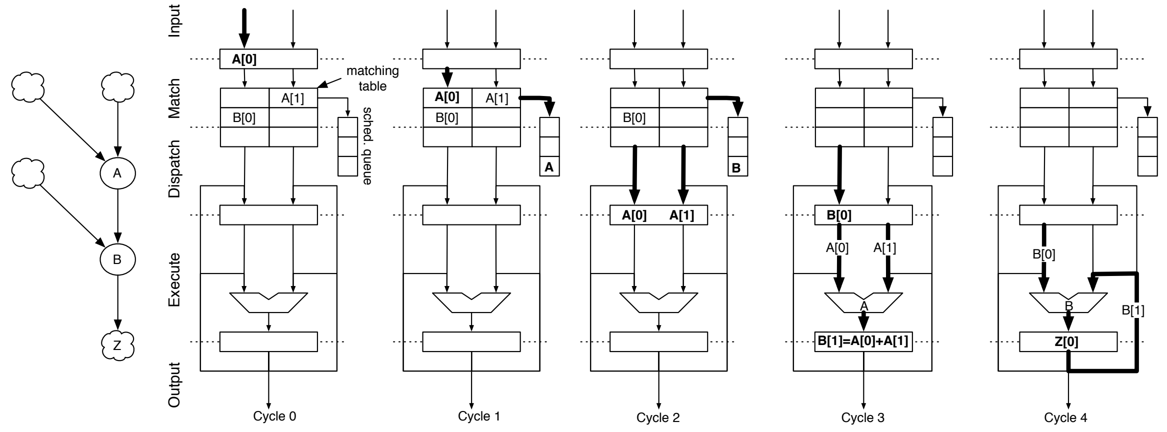


Figure 4.3: **The flow of operands through the PE pipeline and forwarding networks:** A PE executes a simple series of instructions.

bypass network allows instructions to execute on consecutive cycles. In the diagram, $X[n]$ is the nth input to instruction X. Five consecutive cycles are depicted; before the first of these, one input for each of instructions A and B has arrived and reside in the matching table. The "clouds" in the dataflow graph represent operands that were computed by instructions at other processing elements and have arrived via the input network.

Cycle 0: (at left in Figure 4.3) Operand $A[0]$ arrives and INPUT accepts it.

Cycle 1: MATCH writes $A[0]$ into the matching table and, because both its inputs are present, places A into the scheduling queue.

Cycle 2: DISPATCH chooses A for execution and reads its operands from the matching table. At the same time, it recognizes that A's output is destined for B. In preparation for this producer-consumer handoff it inserts B into the scheduling queue.

Cycle 3: DISPATCH reads $B[0]$ from the matching table. EXECUTE computes the result of A, which becomes $B[1]$.

Cycle 4: EXECUTE computes the result of instruction B, using $B[0]$ from DISPATCH

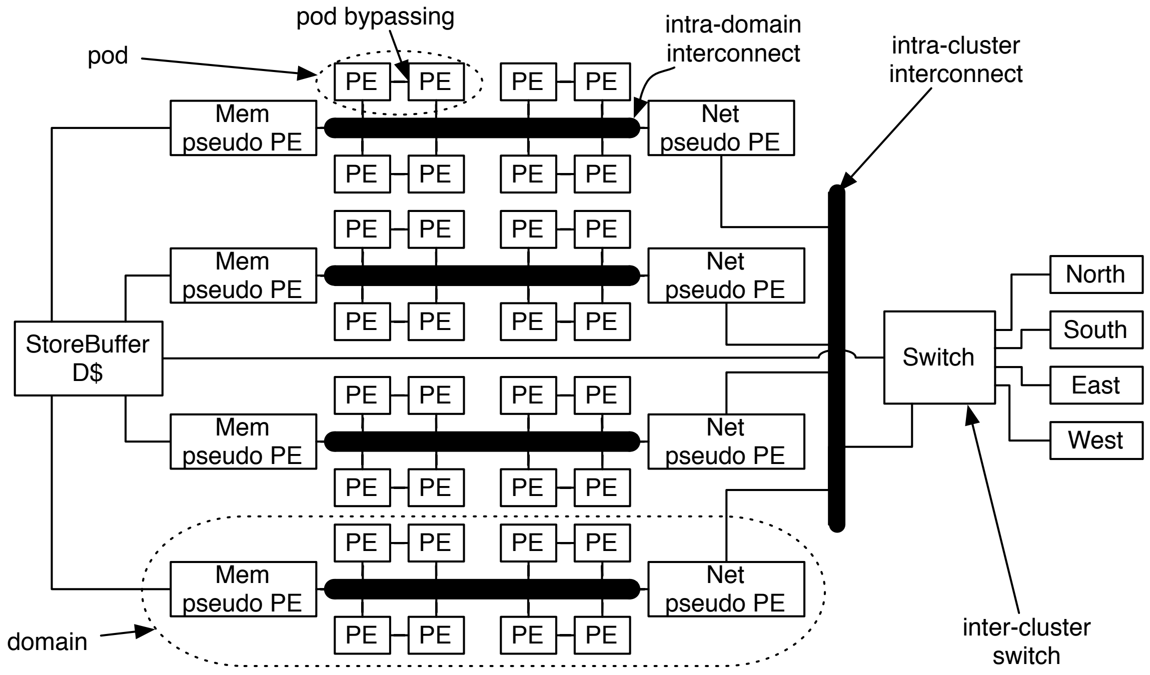


Figure 4.4: **The cluster interconnects:** A high-level picture of a cluster illustrating the interconnect organization.

and $B[1]$ from the bypass network.

Cycle 5 (not shown): Output will send B's result to instruction Z.

The logic in Match and Dispatch is the most complex part of the entire WaveCache architecture, and most of it is devoted to allowing back-to-back execution of dependent instructions while achieving our cycle time goal (22FO4).

4.3 The WaveCache interconnect

The previous section described the execution resource of the WaveCache, the PE. This section will detail how PEs on the same chip communicate. PEs send and receive data using a hierarchical, on-chip interconnect (Figure 4.4). The figure shows the four levels in this hierarchy: intra-pod, intra-domain, intra-cluster and inter-cluster. While the purpose of each network is the same – transmission of instruction operands and memory values – their designs vary significantly. We will describe the salient features of these networks in the next four sections.

4.3.1 PEs in a pod

The first level of interconnect, the intra-pod interconnect, enables two PEs to share scheduling hints and computed results. Merging a pair of PEs into a pod consequently provides lower latency communication between them than using the intra-domain interconnect (described below). Although PEs in a pod snoop each others bypass networks, the rest of their hardware remains partitioned. In particular, they have separate matching tables, scheduling and output queues, and ALUs.

The decision to integrate pairs of PEs together is a response to two competing concerns: we want the clock cycle to be short *and* instruction-to-instruction communication to take as few cycles as possible. To reach our cycle-time goal, the PE and the intra-domain interconnect (described next) need to be pipelined. This increases average communication latency and reduces performance significantly. Allowing pairs of PEs to communicate quickly brings the average latency back down without significantly impacting cycle time. Tightly integrating more PEs would increase complexity significantly, and our data showed that the gains in performance are small.

4.3.2 The intra-domain interconnect

PEs communicate with PEs in other pods over an intra-domain interconnect. In addition to the eight PEs in the domain, the intra-domain interconnect also connects two *pseudo-PEs* that serve as gateways to the memory system (the Mem pseudo-PE) and the other PEs on the chip (the Net pseudo-PE). The pseudo-PEs' interface to the intra-domain network is identical to that of a normal PE.

The intra-domain interconnect is broadcast-based. Each of the eight PEs has a dedicated result bus that carries a single data result to the other PEs in its domain. Each pseudo-PE also has a dedicated output bus. PEs and pseudo-PEs communicate over the intra-domain network using a garden variety Ack/Nack network.

4.3.3 The intra-cluster interconnect

The intra-cluster interconnect provides communication between the four domains' Net pseudo-PEs. It also uses a Ack/Nack network similar to that of the intra-domain interconnect.

4.3.4 The inter-cluster interconnect

The inter-cluster interconnect is responsible for all long-distance communication in the WaveCache. This includes operands traveling between PEs in distant clusters and coherence traffic for the L1 caches.

Each cluster contains an inter-cluster network switch, that routes messages between six input/output ports: four of the ports lead to the network switches in the four cardinal directions, one is shared among the four domains' Net pseudo-PEs, and one is dedicated to the store buffer and L1 data cache.

Each input/output port supports the transmission of up to two operands. Its routing follows a simple protocol: the current buffer storage state (i.e., whether space is available or not) for each cardinal direction at each switch is sent to the adjacent switch in that direction. The adjacent switches receive this information a clock cycle later, and will only send data if the receiver is guaranteed to have space.

The inter-cluster switch provides two virtual channels that the interconnect uses to prevent deadlock [19]. Each output port contains two 8-entry output queues (one for each virtual network). In some cases, a message may have two possible directions (e.g., North and West if its ultimate destination is to the northwest). In these cases the router randomly selects which way to route the message.

4.4 The store buffer

The hardware support for wave-ordered memory lies in the WaveCache's store buffers. The store buffers, one per cluster, are responsible for implementing the wave-ordered

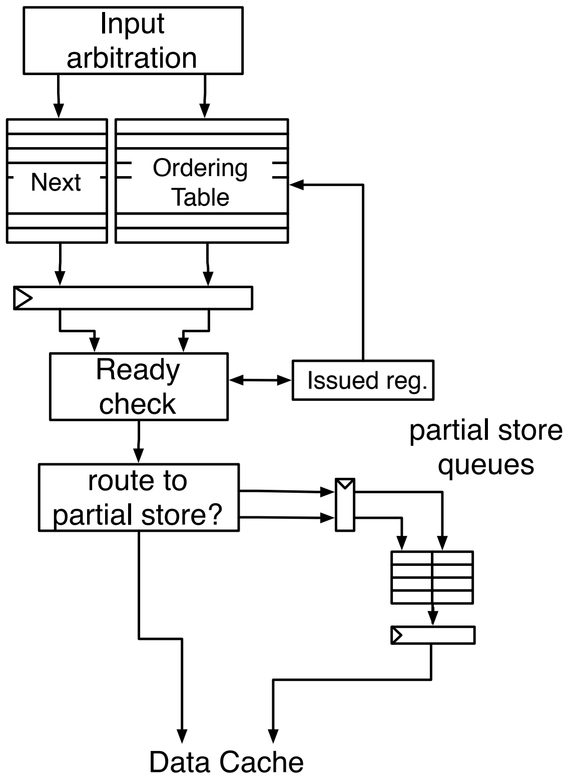


Figure 4.5: **The store buffer:** The microarchitecture of the store buffer.

memory interface that guarantees correct memory ordering. To access memory, processing elements send requests to their local store buffer via the MEM pseudo-PE in their domain. The store buffer will either process the request or direct it to another store buffer via the inter-cluster interconnect (see below). All memory requests for a single *dynamic* instance of a wave (for example, an iteration of an inner loop), including requests from both local and remote processing elements, are managed by the same store buffer.

To simplify the description of the store buffer's operation, we denote pred(W), seq(W), and succ(W) as the wave-ordering annotations for a request W. We also define next(W) to be the sequence number of the operation that actually follows W in the current instance of the wave. next(W) is determined either directly from succ(W) or is calculated by the wave-ordering hardware, if succ(W) is '?'.

The store buffer (Figure 4.5) contains four major microarchitectural components: an ORDERING table, a NEXT table, an ISSUED register, and a collection of *partial store queues*. Store buffer requests are processed in three pipeline stages: MEMORY-INPUT writes newly-arrived requests into the ORDERING and NEXT tables. MEMORY-SCHEDULE reads up to four requests from the ORDERING table and checks to see if they are ready to issue. MEMORY-OUTPUT dispatches memory operations that can issue to the cache or to a partial store queue (described below). We detail each pipeline stage of this memory interface below.

MEMORY-INPUT accepts up to four new memory requests per cycle. It writes the address, operation and data (if available in the case of STOREs) into the ORDERING table at the index seq(W). If succ(W) is defined (i.e., not '?'), the entry in the NEXT table at location seq(W) is updated to succ(W). If pred(W) is defined, the entry in the NEXT table at location pred(W) is set to seq(W).

MEMORY-SCHEDULE maintains the ISSUED register, which points to the next memory operations to be dispatched to the data cache. It uses this register to read four entries from the NEXT and ORDERING tables. If any memory ordering links can be formed (i.e., NEXT table entries are not empty), the memory operations are dispatched to MEMORY-OUTPUT and the ISSUED register is advanced. The store buffer supports the decoupling of store-data from store addresses as described in Section 3.4.4. This is done with a hardware structure called a *partial store queue*, described below. The salient point for MEMORY-SCHEDULE, however, is that STOREs are sent to MEMORY-OUTPUT even if their data has not yet arrived.

Partial store queues (see Section 3.4.4) take advantage of the fact that store addresses can arrive significantly before their data. In these cases, a partial store queue stores all operations to the same address. These operations must wait for the data to arrive, but other operations may proceed. When the data finally arrives all, the operations in the partial store queue can be applied in quick succession. The store buffer contains two partial store queues.

MEMORY-OUTPUT reads and processes dispatched memory operations. Four situations can occur. (1) The operation is a LOAD or a STORE with its data is present: The operation proceeds to the data cache. (2) The operation is a LOAD or a STORE and a partial store queue exists for its address: The memory operation is sent to the partial store queue. (3) The memory operation is a STORE, its data has not yet arrived, and no partial store queue exists for its address: A free partial store queue is allocated and the STORE is sent to it. (4) The operation is a LOAD or a STORE, but no free partial store queue is available or the partial store queue is full: The operation is discarded and the ISSUED register is rolled back. The operation will reissue later.

4.4.1 Routing memory requests

A single store buffer must handle all the memory requests for a single wave, but the memory instructions that produce the requests might execute anywhere in the array of processing elements. To ensure that requests find their way to the correct store buffer, the WaveScalar hardware maintains a map of wave numbers to store buffers.

When a memory instruction fires, it sends its request to the store buffer in the local cluster. The store buffer checks the map, which is stored in main memory, and takes one of three actions: (1) If the request is destined for the local store buffer, it processes it. (2) If it is destined for the store buffer in another cluster, it forwards it there. (3) If the map shows that no store buffer is currently handling the wave, it takes on the responsibility for the wave and updates the map accordingly using a read-modify-write operation. Since the map is stored in main memory, the coherence protocol ensures that all store buffers see a consistent view.

When a wave completes, the store buffer checks the map again to find the store buffer for the next wave in the program. It sends a message to that store buffer indicating that all previous memory operations are complete. Because execution for a single thread tends to be localized at a single cluster, reading and updating the map requires minimal coherence traffic. Likewise, the messages between store buffers are usually local (i.e., the local store buffer is handling the next wave as well).

4.5 Caches

The rest of the WaveCache's memory hierarchy comprises a four-way set associative L1 data cache at each cluster, and an L2 cache distributed along the edge of the chip. A directory-based, multiple readers, single writer coherence protocol keeps the L1 caches consistent. All coherence traffic travels over the inter-cluster interconnect.

The L1 data cache has a 3-cycle hit delay. The L2's hit delay is 14-30 cycles depending upon the address and the distance to the requesting cluster. Main memory latency is modeled at 200 cycles.

4.6 Placement

Placing instructions carefully into the WaveCache is critical to good performance. Instructions' proximity determines the communication latency between them, arguing for tightly packing instructions together. On the other hand, instructions that can execute simultaneously should reside at different processing elements, because competition for the single functional unit will serialize them.

We continue to investigate the placement problem, and details of our endeavors are available in [43]. Here, we describe the approach we used for the studies in this thesis.

The placement scheme has a static and a dynamic component. At compile time, the compiler performs a pre-order, depth-first traversal of the dataflow graph of each function to generate a linear ordering of the instructions. We chose this traversal, because it tends to make chains of dependent instructions in the dataflow graph appear consecutively in the ordering. The compiler breaks the sequence of instructions into short segments. We tune the segment length for each application.

At runtime, the WaveCache loads these short segments of instructions when an instruction in a segment that is not mapped into the WaveCache needs to execute. The entire segment is mapped to a single PE. Because of the ordering the compiler used to generate the segments, they will usually be dependent on one another. As a result, they will not compete for execution resources, but instead will execute on consecutive cycles. The algorithm fills all the PEs in a domain, and then all the domains in a cluster, before moving on to the next cluster. It fills clusters by “snaking” across the grid, moving from left to right on the even rows and right to left on the odd rows.

4.7 Managing parallelism

Our placement scheme does a good job of scheduling execution and communication resources, but a third factor, the so-called “parallelism explosion”, can have a strong effect on performance in dataflow systems. Parallelism explosion occurs when part of an application (e.g., the index computation of an inner loop) runs ahead of the rest of program, generating a vast number of tokens that will not be consumed for a long time. These tokens overflow the matching table and degrade performance as they spill to memory. We use a well-known dataflow technique, k-loop bounding [15], to restrict the number iterations that can be executing at one time to k. We tune k for each application.

Chapter 5
RUNNING MULTIPLE THREADS IN WAVESCALAR

The WaveScalar architecture described so far can support a single executing thread. Modern applications, such as databases and web servers, use multiple threads both as a useful programming abstraction and to increase performance by exposing parallelism.

Recently, manufacturers have begun placing several processors on a single die to create chip multiprocessors (CMPs). There are two reasons for this move: First, scaling challenges will make designing ever-larger superscalar processors infeasible. Second, commercial workloads are often more concerned with the aggregate performance of many threads rather than single-thread performance. Any architecture intended as an alternative to CMPs must be able to execute multiple threads simultaneously.

This section extends the single-threaded WaveScalar design to execute multiple threads. The key issues we must address to allow WaveScalar to execute multiple threads are managing multiple, parallel sequences of wave-ordered memory operations, differentiating between data values that belong to different threads, and allowing threads to communicate. WaveScalar's solution to these problems are all simple and efficient. For instance, WaveScalar is the first architecture to allow programs to manage memory ordering directly by creating and destroying memory orderings and dynamically binding them to a particular thread. WaveScalar's thread-spawning facility is efficient enough to parallelize small loops. Its synchronization mechanism is also light-weight and tightly integrated into the dataflow framework.

The required changes to the WaveCache to support the ISA extensions are surprisingly small, and they do not impact the overall structure of the WaveCache, because executing threads dynamically share most WaveCache processing resources.

The next two sections describe the multihthreading ISA extensions, then we discuss the changes needed to the hardware.

5.1 Multiple memory orderings

As previously introduced, the wave-ordered memory interface provides support for a single memory ordering. Forcing all threads to contend for the same memory interface, even if it were possible, would be detrimental to performance. Consequently, to support multiple threads, we extend the WaveScalar instruction set to allow multiple independent sequences of ordered memory accesses, each of which belongs to a separate thread. First, we annotate every data value with a Thread-Id in addition to its Wave-Number. Then, we introduce instructions to associate memory-ordering resources with particular Thread-Ids.

Thread-Ids The WaveCache already has a mechanism for distinguishing values and memory requests within a single thread from one another – they are tagged with Wave-Numbers. To differentiate values from different threads, we extend this tag with a Thread-Id and modify WaveScalar's dataflow firing rule to require that operand tags match on both Thread-Id and Wave-Number. As with Wave-Numbers, additional instructions are provided to directly manipulate Thread-Ids. In figures and examples throughout the rest of this thesis, the notation $<t, w>.d$ signifies a token tagged with Thread-Id t and Wave-Number w and having data value d.

To manipulate Thread-Ids and Wave-Numbers, we introduce several instructions that convert Wave-Numbers and Thread-Ids to normal data values and back again. The most powerful of these is Data-To-Thread-Wave, which sets both the Thread-Id and Wave-Number at once; Data-To-Thread-Wave takes three inputs, $<t_0, w_0>.t_1$, $<t_0, w_0>.w_1$, and $<t_0, w_0>.d$ and produces as output

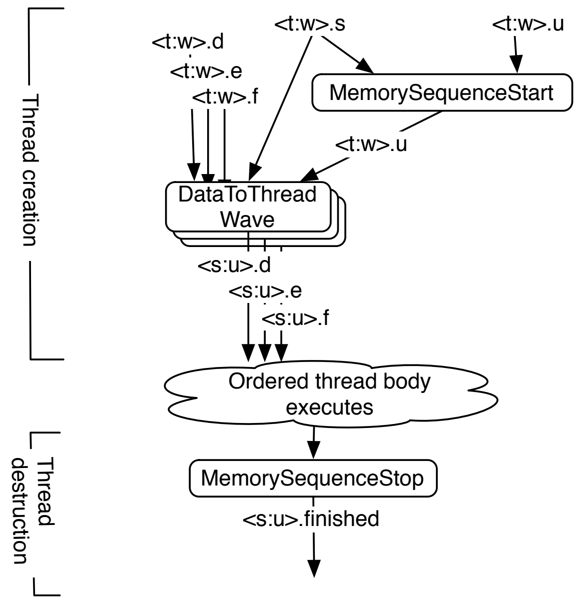


Figure 5.1: **Thread creation and destruction:** MEMORY-SEQUENCE-START and MEMORY-SEQUENCE-STOP in action.

$<t_1, w_1>.d$. WaveScalar also provides two instructions (DATA-TO-THREAD and DATA-TO-WAVE) to set THREAD-IDS and WAVE-NUMBERS separately, as well as two instructions (THREAD-TO-DATA and WAVE-TO-DATA) to extract THREAD-IDS and WAVE-NUMBERS. Together, all these instructions place WaveScalar's tagging mechanism completely under programmer control, and allow programmers to write software such as threading libraries. For instance, when the library spawns a new thread, it must relabel the inputs with the new thread's THREAD-ID and the WAVE-NUMBER of the first wave in its execution. DATA-TO-THREAD-WAVE accomplishes exactly this task.

Managing memory orderings: Having associated a THREAD-ID with each value and memory request, we now extend the wave-ordered memory interface to enable programs to associate memory orderings with THREAD-IDS. Two new instructions control the creation and destruction of memory orderings, in essence creating and terminating coarse-grain threads: MEMORY-SEQUENCE-START and MEMORY-SEQUENCE-

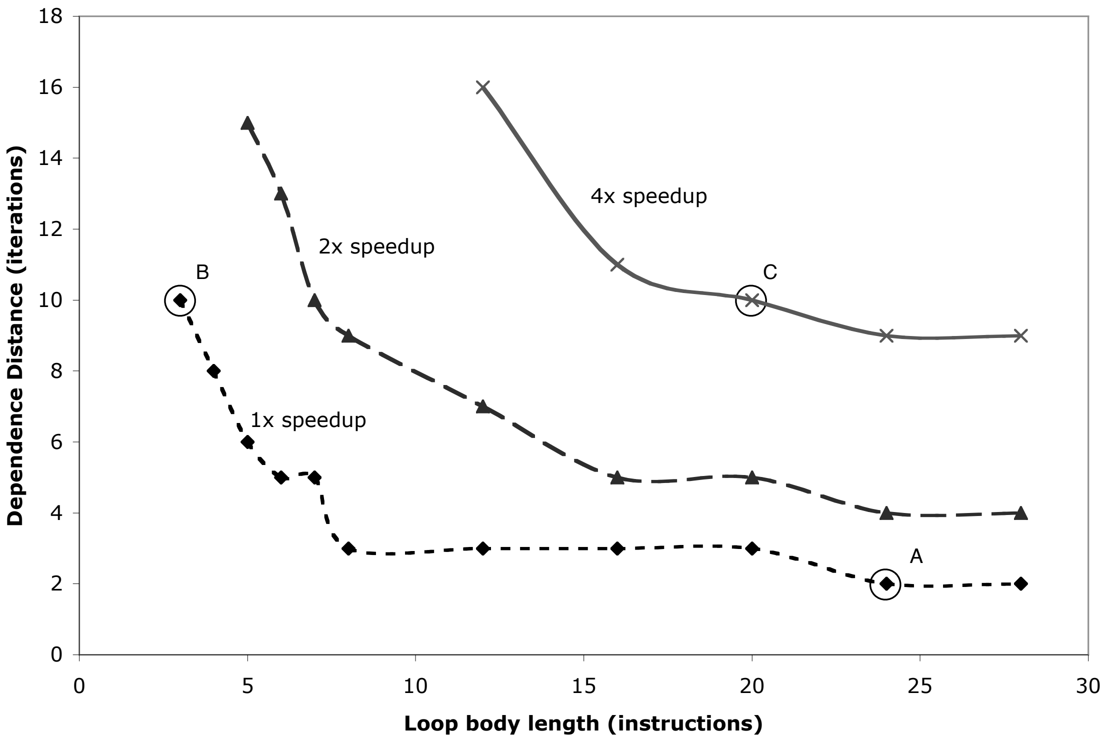


Figure 5.2: **Thread creation overhead:** Contour plot of speed up possible for various thread granularities

STOP.

MEMORY-SEQUENCE-START creates a new wave-ordered memory sequence for a new thread. This sequence is assigned to a store buffer, which services all memory requests tagged with the thread's THREAD-ID and WAVE-NUMBER; requests with the same THREAD-ID but a different WAVE-NUMBER cause a new store buffer to be allocated.

MEMORY-SEQUENCE-STOP terminates a memory ordering sequence. The wave-ordered memory system uses this instruction to ensure that all memory operations in the sequence have completed before its store buffer resources are released.

Figure 5.1 illustrates how to use these two instructions to create and destroy a thread. Thread t spawns a new thread s by sending s's THREAD-ID (s) and WAVE-NUMBER (u) to MEMORY-SEQUENCE-START, which allocates a store buffer to handle the first wave in the new thread. The result of the MEMORY-SEQUENCE-START instruction triggers the three DATA-TO-THREAD-WAVE instructions that set up s's three input parameters. The inputs to each DATA-TO-THREAD-WAVE instruction are a parameter value (d, e, or f), the new THREAD-ID (s) and the new WAVE-NUMBER (u). A token with u is produced by MEMORY-SEQUENCE-START deliberately, to guarantee that no instructions in thread s execute until MEMORY-SEQUENCE-START has finished allocating its store buffer. Thread s terminates with MEMORY-SEQUENCE-STOP, whose output token $<s, u>.finished$ guarantees that its store buffer area has been deallocated.

Implementation Adding support for multiple memory orderings requires only small changes to the WaveCache's microarchitecture. First, the widths of the communication busses and operand queues must be expanded to hold THREAD-IDs. Second, instead of storing each static instruction from the working set of a program in the WaveCache, one copy of each static instruction is stored for each thread. This means that if two threads are executing the same static instructions, each may map the static instructions to different PEs. Finally, the PEs must implement the Thread-Id and Wave-Number manipulation instructions.

Efficiency The overhead associated with spawning a thread directly affects the granularity of extractable parallelism. To assess this overhead in the WaveCache, we designed a controlled experiment consisting of a simple parallel loop in which each iteration executes in a separate thread. The threads have their own wave-ordered memory sequences but do not have private stacks, so they cannot make function calls.

We varied the size of the loop body, which affects the granularity of parallelism, and the dependence distance between memory operands, which affects the number of threads that can execute simultaneously. We then measured speedup compared to a serial execution of a loop doing the same work. The experiment's goal is to answer the following question: Given a loop body with a critical path length of N instructions and a dependence distance that allows T iterations to run in parallel, can the WaveCache speed up execution by spawning a new thread for every loop iteration?

Figure 5.2 is a contour plot of speedup of the loop as a function of its loop size (critical path length in Add instructions, the horizontal axis) and dependence distance (independent iterations, the vertical axis). Contour lines are shown for speedups of $1\times$ (no speedup), $2\times$ and $4\times$. The area above each line is a region of program speedup at or above the labeled value. The data show that the WaveScalar overhead of creating and destroying threads is so low that for loop bodies of only 24 dependent instructions and a dependence distance of 3, it becomes advantageous to spawn a thread to execute each iteration ('A' in the figure). A dependence distance of 10 reduces the size of profitably parallelizable loops to only 4 instructions ('B'). Increasing the number of instructions to 20 quadruples performance ('C').

5.2 Synchronization

The ability to efficiently create and terminate pthread-style threads, as described in the previous section, provides only part of the functionality required to make multi-threading useful. Independent threads must also synchronize and communicate with one another. To this end, WaveScalar provides a memory fence instruction that allows WaveScalar to enforce a relaxed consistency model and a specialized instruction that models a hardware queue lock.

5.2.1 Memory fence

Wave-ordered memory provides a single thread with a consistent view of memory, since it guarantees that the results of earlier memory operations are visible to later operations. In some situations, such as before taking or releasing a lock, a multi-threaded processor must guarantee that the results of a thread's memory operations are visible to *other* threads. We add to the ISA an additional instruction, MEMORY-NOP-ACK that provides this assurance by acting as a memory fence. MEMORY-NOP-ACK prompts the wave-ordered interface to commit the thread's prior loads and stores to memory, thereby ensuring their visibility to other threads and providing WaveScalar with a relaxed consistency model [4]. The interface then returns an acknowledgment, which the thread can use to trigger execution of its subsequent instructions.

5.2.2 Interthread synchronization

Most commercially deployed multiprocessors and multi-threaded processors provide interthread synchronization through the memory system via primitives such as TEST-AND-SET, COMPARE-AND-SWAP, or LOAD-LOCK/STORE-CONDITIONAL. Some research efforts also propose building complete locking mechanisms in hardware [27, 62]. Such queue locks offer many performance advantages in the presence of high lock contention.

In WaveScalar, we add support for queue locks in a way that constrains neither the number of locks nor the number of threads that may contend for the lock. This support is embodied in a synchronization instruction called THREAD-COORDINATE, which synchronizes two threads by passing a value between them. THREAD-COORDINATE is similar in spirit to other lightweight synchronization primitives [34, 11], but is tailored to WaveScalar's execution model.

As Figure 5.3 illustrates, THREAD-COORDINATE requires slightly different matching rules[1]. All WaveScalar instructions *except* THREAD-COORDINATE fire when the tags of two input values match, and they produce outputs with the same tag (Figure 5.3, left). For example, in the figure, both the input tokens and the result have THREAD-ID t_0 and WAVE-NUMBER w_0.

In contrast, THREAD-COORDINATE fires when the *data value* of a token at its first input matches the THREAD-ID of a token at its second input. This is depicted on the right side of Figure 5.3, where the data value of the left input token and the THREAD-ID of the right input token are both t_1. THREAD-COORDINATE generates an output token with the THREAD-ID and WAVE-NUMBER from the first input and the data value from the second input. In Figure 5.3, this produces an output of $<t_0, w_0>.d$. In essence, THREAD-COORDINATE passes the second input's value (d) to the thread of the first input (t_0). Since the two inputs come from different threads, this forces the receiving thread (t_0 in this case) to wait for the data from the sending thread (t_1) before continuing execution.

To support THREAD-COORDINATE in hardware, we augment the tag matching logic at each PE. We add two counters at each PE, one for each input to a THREAD-COORDINATE instruction. When an input arrives, the PE replaces its

[1]Some previous dataflow machines altered the dataflow firing rule for other purposes. For example, Sigma-1 used "sticky" tags to prevent the consumption of loop-invariant data and "error" tokens to swallow values of instructions that incurred exceptions [55]. Monsoon's M-structure store units had a special matching rule to enforce load-store order [51].

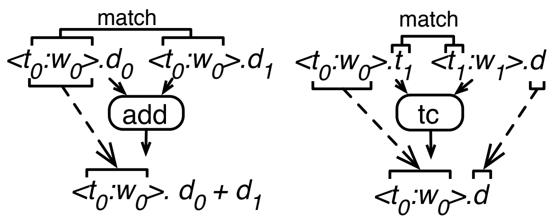


Figure 5.3: **Tag matching:** Matching rules for a normal instruction (left) and a Thread-Coordinate (right).

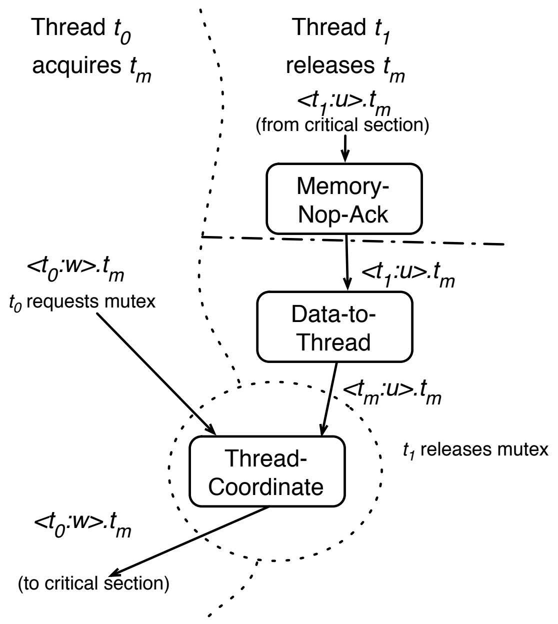


Figure 5.4: **A mutex:** Thread-Coordinate-based mutex.

Wave-Number with the value of the corresponding counter and then increments the counter, ensuring that the tokens are processed in FIFO order. Using this relabeling, the matching queues naturally form a serializing queue with efficient constant time access and no starvation.

Although one can construct many kinds of synchronization objects using Thread-Coordinate, we only illustrate a simple mutex (Figure 5.4) here. In this case, Thread-Coordinate is the vehicle by which a thread releasing a mutex passes control to another thread wishing to acquire it.

The mutex in Figure 5.4 is represented by a THREAD-ID, t_m, although it is not a thread in the usual sense; instead, t_m's sole function is to uniquely name the mutex. A thread t_1 that has locked mutex t_m releases it in two steps (right side of figure). First, t_1 ensures that the memory operations it executed inside the critical section have completed by executing MEMORY-NOP-ACK. Then, t_1 uses DATA-TO-THREAD to create the token $<t_m, u>.t_m$, which it sends to the second input port of THREAD-COORDINATE, thereby releasing the mutex.

Another thread, t_0 in the figure, can attempt to acquire the mutex by sending $<t_0, w>.t_m$ (the data is the mutex) to THREAD-COORDINATE. This token will either find the token from t_1 waiting for it (i.e., the lock is free) or await its arrival (i.e., t_1 still holds the lock). When the release token from t_1 and the request token from t_0 are both present, THREAD-COORDINATE will find that they match according to the rules discussed above, and it will then produce a token $<t_0, w>.t_m$. If all instructions in the critical section guarded by mutex t_m depend on this output token (directly or via a chain of data dependences), thread t_0 cannot execute the critical section until THREAD-COORDINATE produces it.

5.3 Discussion

With the addition of coarse-grain multi-threading support, WaveScalar can run any program that a conventional chip multiprocessor can. Unlike a CMP, however, WaveScalar's multithreading support caused very little change to the underlying hardware. Multithreading is simpler on WaveScalar, because the WaveCache architecture is more flexible. Because CMPs use the von Neumann execution model, some resources (e.g., the program counter and register file) must be replicated for each thread, and, as a result, the difference between a single- and multi-threaded processor is great. In WaveScalar's dataflow model, a thread requires no centralized control or resource, so two threads can run almost as easily as one. In fact, the addition of THREAD-IDS and the need for MEMORY-SEQUENCE-START and MEMORY-SEQUENCE-STOP are both results of WaveScalar's need to support von Neumann-style memory semantics. They would not be needed in a pure dataflow system.

The next two chapters describe our experimental infrastructure, evaluate the multithreaded WaveCache, and demonstrate that it achieves both superior efficiency and performance compared to von Neumann processors. Chapter 8 takes WaveScalar beyond the constraints of von Neumann execution and demonstrates the power of its dataflow execution model.

Chapter 6
EXPERIMENTAL INFRASTRUCTURE

This chapter describes the experimental infrastructure we use to evaluate the WaveScalar architecture described in Chapters 7 and 8. There are three key tools: A synthesizable RTL model and the accompanying synthesis tools, a WaveScalar tool chain that includes a cycle-level architectural simulator, and a set of workloads that span a wide range of different applications.

In then next two chapters we will use these tools to search the space of WaveScalar processor designs and compare the best designs to contemporary single-threaded and multi-threaded processors. We describe each of the tools in turn.

6.1 The RTL model

Since WaveScalar is a tiled dataflow processor, it is different enough from conventional von Neumann processors that we cannot draw on past research, existing tools, or industrial experience to understand area and cycle-time requirements. Since these parameters are crucial for determining how well WaveScalar performs and how to partition the silicon resources, we constructed a synthesizable RTL model of the components described in Chapters 4 and 5.

The synthesizable RTL model is written in Verilog, and targets a 90nm ASIC implementation. Considerable effort was put into designing, and redesigning this Verilog to be both area-efficient and fast. The final clock speed (22 FO4) comes from our fourth major redesign.

The 90nm process is the most current process technology available, so the results presented here should scale well to future technologies. In addition to using a modern process, we performed both front-end and back-end synthesis to get as realistic a model as possible. The model makes extensive use of Synopsys DesignWare IP [2] for critical components such as SRAM controllers, queue controllers, arbiters, and arithmetic units. The design currently uses a single frequency domain and a single voltage domain, but the tiled and hierarchical architecture would lend itself easily to multiple voltage and frequency domains in the future.

6.1.1 ASIC design flow

We used the most up-to-date tools available for Verilog synthesis. Synopsys VCS provided RTL simulation and functional verification of the post-synthesis netlists. Front-end synthesis was done using Synopsys DesignCompiler. Cadence FirstEncounter handled back-end synthesis tasks such as floorplanning, clock-tree synthesis, and place and route [1]. By using back-end synthesis, the area and timing results presented here include realistic physical effects, such as incomplete core utilization and wire delay, that are critical for characterizing design performance at 90nm and below.

6.1.2 Standard cell libraries

Our design uses the 90nm high-performance GT standard cell libraries from Taiwan Semiconductor Manufacturing Company (TSMC) [3]. The library contains three implementations of cells, each with a different threshold voltage, for balancing power and speed. We allow DesignCompiler and FirstEncounter to pick the appropriate cell implementation for each path.

The memory in our design is a mixture of SRAM memories generated from a commercial memory compiler (used for the large memory structures, such as data caches) and Synopsys DesignWare IP memory building blocks (used for smaller memory structures).

6.1.3 Timing data

Architects commonly measure clock cycle time in a process-independent metric, *fanout-of-four* (FO4). A design's cycle time in FO4 does not change (much) as the fabrication process changes, thus enabling a more direct comparison of designs across process technologies.

Synthesis tools, however, report delay in absolute terms (nanoseconds). To convert nanoseconds to FO4, we followed academic precedent [61] and used the technique suggested in [14] to measure the absolute delay of one FO4. We synthesized a ring oscillator using the same design flow and top-speed standard cells (LVT) used in our design and measured FO1 (13.8ps). We then multiplied this delay by three to yield an approximation of one FO4 (41.4ps). All timing data presented here is reported in FO4 based upon this measurement.

6.2 The WaveScalar tool chain

We use a binary translator-based tool chain to create WaveScalar executables. Figure 6.1 shows the components of the system and how they interact. Three components, the binary translator, the performance simulator, and the placement system, are most important. We address the translator and simulator here. Section 4.6 describe the placement algorithm we use.

6.2.1 The binary translator

The binary translator converts Alpha AXP binaries into WaveScalar assembly language. There are two key tasks the translator must accomplish: First, it converts the von Neumann style control structures (branches, PC-based function calls, etc.) into dataflow constructs. Second, it breaks the program into waves and adds annotations that allow the wave-ordered memory system to enforce the correct memory ordering.

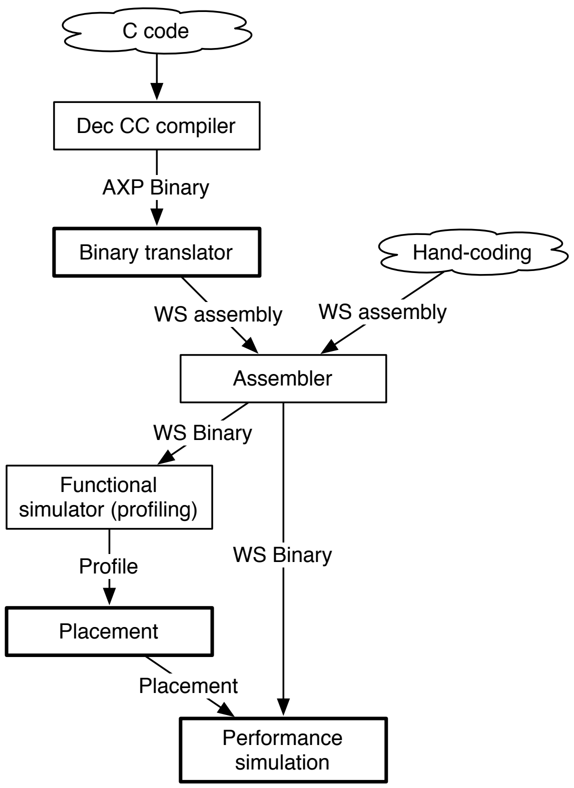


Figure 6.1: **The WaveScalar tool chain:** The tool chain we use to compile and execute WaveScalar programs.

Control

In von Neumann systems, the program counter controls execution. Branches and function calls manipulate the program counter to steer execution along the correct path in the program. WaveScalar uses the STEER instruction (see Chapter 2) to guide execution instead.

Replacing PC-based conditional branch instructions with STEER instructions requires inserting one STEER instruction for each of the 64 registers the Alpha ISA defines at each branch point. This results in a very large number of STEER instructions, and the binary translator uses several techniques, including control dependence analysis and dead value analysis, to optimize away as many of them as possible.

The binary translator uses the INDIRECT-SEND instruction to implement register-indirect jumps and function calls as described in Chapter 2.

Wave creation

To support wave-ordered memory (Chapter 3), the translator must break each functions into a series of waves. Chapter 3 defines precisely the properties that a wave must have, but the binary translator generates waves that satisfy a simpler set of properties: It produces waves that have a single entrance, only contain function calls at wave exits, and contain no back edges.

To break up a function, the translator uses control analysis to label back edges in the control flow graph. Then it uses the entrance block of the function as the entrance of the first wave. The translator expands that wave using a breadth-first traversal of the control flow graph. It stops the wave's growth along a path when it encounters a function call or a basic block with a back edge entering or leaving it. Once the first wave has stopped growing, the translator selects one of the points at which it stopped and starts a new wave with that basic block.

To assign predecessor, successor, sequence, and ripple numbers, the translator traverses each wave in depth-first order and assigns or updates those annotations as necessary.

Limitations

The binary translator was written for two reasons: First it provided an expedient path to generating a WaveScalar executable from C code. Second, it demonstrates that building a von Neumann-to-dataflow translator is possible, laying the groundwork for a smoother transition between the two execution models.

The approach suffers from several limitations, however. First, using Alpha binaries as the input to the translator severely restricts the optimizations that the binary translator can perform. For instance, alias analysis is effectively impossible, because no high-level information about memory use is available. The DEC compiler partially compensates for this limitation, since it provides good control over many optimizations that are useful in WaveScalar compilation (e.g., function inlining and loop unrolling).

Second, the binary translator cannot handle irreducible loops (usually the result of `goto`s), jump-table-based `switch` statements, or compiler intrinsics that do not follow the Alpha calling convention. These limitations are strictly a deficiency of our translator, not a limitation of WaveScalar's ISA or execution model. In practice, we restructure the source code to eliminate these constructs, if possible.

Our group is currently working on a full-fledged WaveScalar compiler that will address all of these shortcomings and provide the foundation a more detailed exploration of WaveScalar compilation.

6.2.2 The performance simulator

To measure WaveScalar performance we use a cycle-level processor simulator. The simulator is very detailed and models all the components of the architecture including the pipelined PE, the wave-ordered store buffer, the network switch, the memory hierarchy, and all the interconnects that connect these components. It also answers basic questions, such as how the sizing of microarchitectural features affect performance. We use a set of microbenchmarks to check that the simulator's timing and behavior are correct along with a set of larger regression tests that ensure application-level correctness.

The simulator also provides emulation for the operating system by providing implementations of the necessary system calls. It also emulates some features that a WaveScalar operating system would provide. For instance, the simulator handles the dynamic portion of the instruction placement process and accounts for an approximation of the delay it would incur.

To make measurements comparable with conventional architectures, we measure performance in *alpha instructions per cycle* (AIPC) and base our superscalar comparison on a machine with similar clock speed [32]. AIPC measures the number of non-overhead instructions (i.e., instructions other than STEER, ϕ, etc.) executed per cycle. The AIPC measurements for the superscalar architectures we compare to are in good agreement with other measurements [45].

6.3 Applications

We used three groups of workloads to evaluate the WaveScalar processor. Each focuses on a different aspect of WaveScalar performance. To measure single-threaded performance, we chose a selection of the Spec2000 [58] benchmark suite (*ammp*, *art*, *equake*, *gzip*, *twolf* and *mcf*). To evaluate the processor's media processing performance we use *rawdaudio*, *mgeg2encode*, and *djpeg* from Mediabench [38]. Finally, we use six of the Splash2 [6] benchmarks, *fft*, *lu-continuous*, *ocean-noncontinuous*, *raytrace*, *water-spatial*, and *radix*, to explore multi-threaded performance.

We compiled each application with the DEC cc compiler using `-O4 -fast -inline speed` optimizations. A binary translator-based tool chain was used to convert these binaries into WaveScalar assembly and then into WaveScalar binaries. The choice of benchmarks represents a range of applications as well as the limitations of our

Table 6.1: **Workload configurations:** Workloads and parameters used in this thesis.

	Benchmark	**Parameters**
Splash2	fft	-m12
	lu	-n128
	radix	-n16384 -r32
	ocean-noncont	-n18
	water-spatial	64 molecules
	raytrace	-m64 car.env
MediaBench	mpeg	options.par data/ out.mpg
	djpeg	-dct int -ppm -outfile testout.ppm testorig.jpg
	adpcm	< clinton.adpcm
SpecInt	gzip	/ref/input/input.source 60
	twolf	ref/input/ref
	mcf	ref/input/inp.in
SpecFP	ammp	< ref/input/ammp.in
	art	-scanfile ref/input/c756hel.in -trainfile1 ref/input/a10.img -trainfile2 ref/input/hc.img -stride 2 -startx 470 -starty 140 -endx 520 -endy 180 -objects 10
	equake	< ref/input/inp.in

binary translator (see above). Table 6.1 shows the configuration parameters for the workloads. We skip past the initialization phases of all our workloads.

Chapter 7

WAVECACHE PERFORMANCE

Chapters 2-5 described WaveScalar's single- and multi-threaded instruction set and presented the WaveCache processor design. The WaveCache design defines a spectrum of processor configurations. At one end of the spectrum is a small WaveCache processor, comprising just a single cluster, which would be suitable for small, single-threaded or embedded applications. At the other end, a supercomputer processor might contain tens of clusters and hundreds or thousands of processing elements. The ability to move easily along this design continuum is a key objective of tiled architectures. A second objective is that they be able to tolerate such drastic changes in area by localizing data communication, thereby reducing latency. This section explores how well the WaveScalar architecture performs while achieving these goals.

We begin in Section 7.1 with a detailed look at the area budget of a particular WaveScalar processor configuration. Then, to understand a larger design space, Section 7.2 uses data from our RTL synthesis to develop an area model that describes the area requirements for a range of designs. We use the resulting model to enumerate a large class of WaveScalar processor designs that could be built in modern process technology. We evaluate these designs using our suite of single- and multi-threaded workloads and use the results to perform an area/performance pareto analysis of the WaveScalar design space covered by our RTL design and area model. Lastly, in Section 7.3, we examine changes in the network traffic patterns as the size of the WaveScalar processor increases.

Table 7.1: **A cluster's area budget:** A breakdown of the area required for a single cluster.

	Area in PE	Area in Domain	Area in Cluster	% of PE	% of Domain	% of Cluster
PE						
INPUT	0.01mm^2	0.09mm^2	0.37mm^2	1.2%	1.1%	0.9%
MATCH	0.58mm^2	4.60mm^2	18.41mm^2	61.0%	55.2%	43.3%
DISPATCH	0.01mm^2	0.05mm^2	0.18mm^2	0.6%	0.6%	0.4%
EXECUTE	0.02mm^2	0.19mm^2	0.77mm^2	2.5%	2.3%	1.8%
OUTPUT	0.02mm^2	0.14mm^2	0.55mm^2	1.8%	1.7%	1.3%
instruction store	0.31mm^2	2.47mm^2	9.88mm^2	32.8%	29.6%	23.2%
total	0.94mm^2	7.54mm^2	30.16mm^2	100%	90.5%	71.0%
Domain						
MemPE		0.13mm^2	0.53mm^2		1.6%	1.2%
NetPE		0.13mm^2	0.53mm^2		1.6%	1.2%
8×PE		7.54mm^2	30.16mm^2		90.5%	71.0%
FPU		0.53mm^2	2.11mm^2		6.3%	5.0%
total		8.33mm^2	33.32mm^2		100%	78.4%
Cluster						
4× domain			33.32mm^2			78.4%
network switch			0.37mm^2			0.9%
store buffer			2.62mm^2			6.2%
data cache			6.18mm^2			14.5%
total			42.50mm^2			100.0%

7.1 Area model and timing results

Our RTL toolchain provides both area and delay values for each component of the WaveScalar processor. For this study, we restrict ourselves to processors that achieve a clock rate of 22 FO4, which occurs for a wide range of designs in our Verilog model. This is the shortest cycle time allowed by the critical path within the PE. For most configurations, the critical path is through the execution unit when using operands from the other PE in the pod. However, enlarging the matching cache or instruction cache memory structures beyond 256 entries makes paths in the MATCH and DISPATCH stages critical paths. Floating point units are pipelined to avoid putting floating-point execution on the critical path. INPUT and OUTPUT devote 9 and 5 FO4, respectively, to traversing the intra-domain network, so there is no need for an additional stage for intra-domain wire delay.

Since the critical path is the ALU for designs with smaller than 256-entry matching caches and 256-entry instruction caches, we can resize these structures downward for optimum area-performance without dramatically altering the cycle time. This allows us to evaluate a large number of potential processing element designs based on area without worrying about an accompanying change in cycle time. We confirmed this by synthesizing designs with 16- to 256-entry matching caches and with 8- to 256-entry instruction caches. The clock cycle for these configurations changed by less than 5% until the structures reached 256 entries, at which point the cycle time increased by about 21% for the matching cache and 7% for the instruction cache. These latencies and structure size limits for our study are summarized in Table 7.2 and Table 7.3, respectively.

Table 7.1 shows how the die area is spent for the baseline design described in Table 7.2. Note that the 71% of the cluster area is devoted to PEs. Also, almost all the area, ~83%, is spent on SRAM cells which make up the instruction stores, matching caches, store buffer ordering tables, and L1 data caches.

Table 7.2: **Microarchitectural parameters:** The configuration of the baseline WaveScalar processor

WaveScalar Capacity	4K static instructions (128 per PE)		
PEs per Domain	8 (4 pods)	Domains / Cluster	4
PE Input Queue	128 entries, 4 banks	Network Latency	within Pod: 1 cycle within Domain: 5 cycles within Cluster: 9 cycles inter-Cluster: 9 + cluster dist.
PE Output Queue	4 entries, 2 ports (1r, 1w)		
PE Pipeline Depth	5 stages		
L1 Cache	32KB, 4-way set associative, 128B line, 4 accesses per cycle	Network Switch	2-port, bidirectional
Main RAM	200 cycle latency		

7.2 *Performance analysis*

To understand the performance characteristics of our WaveScalar processor design, we perform a systematic search of the entire design space. Measurements from our RTL model provide an area model for WaveScalar processors. For both single- and multi-threaded workloads, we combine the output of the area model with results from our WaveCache simulator to perform an area-vs-performance pareto analysis of the WaveScalar design space. Then, we compare the performance of several pareto-optimal designs to more conventional von Neumann processors in terms of raw performance and area-efficiency.

7.2.1 The area model

Our area model considers the seven parameters with the strongest effect on the area requirements. Table 7.3 summarizes these parameters (top half of the table) and how they combine with data from the RTL model to form the total area, WC_{area} (bottom of the table). For both the matching table and instruction store, we synthesized versions from 8 to 128 entries to confirm that area varies linearly with capacity. For the L1 and L2 caches, we used the area of 1KB and 1MB arrays provided by a memory compiler to perform a similar verification. The "Utilization factor" is the measure of how densely the tools managed to pack cells together, while still having space for routing. Multiplying by its inverse accounts for the wiring costs in the entire design.

The area model ignores some minor effects. For instance, it assumes that wiring costs do not decrease with fewer than four domains in a cluster, thereby overestimating this cost for small clusters. Nevertheless, the structures accounting for most of the silicon area ($\sim$80% as discussed in Section 7.1) are almost exactly represented.

There are three other WaveScalar parameters that affect performance that are not included in the model because they do not have a large effect on area. These are the widths of the three levels of WaveScalar interconnect. For this study, we allow one

Table 7.3: **WaveScalar processor area model:** Values and expressions used to compute WaveScalar processor area requirements.

Parameter	Symbol	Description	Range
Clusters	C	Number of clusters in the WaveScalar processor	$1 \ldots 64$
Domains/cluster	D	Number of domains per cluster	$1 \ldots 4$
PEs/domain	P	Number of PEs per domain	$2 \ldots 8$
PE virtualization degree	V	Instructions capacity of each PE	$8 \ldots 256$
Matching table entries	M	Number of matching table entries	$16 \ldots 128$
L1 Cache size	$L1$	KB of L1 cache/cluster	$8 \ldots 32$
L2 Cache size	$L2$	total MB of L2 cache	$0 \ldots 32$
Area component	Symbol	Value	
PE matching table	M_{area}	$= 0.004\text{mm}^2/\text{entry}$	
PE instruction store	V_{area}	$= 0.002\text{mm}^2/\text{instruction}$	
Other PE components	e_{area}	$= 0.05\text{mm}^2$	
Total PE	PE_{area}	$= M \times M_{\text{area}} + V \times V_{\text{area}} + e_{\text{area}}$	
Pseudo-PE	PPE_{area}	$= 0.1236\text{mm}^2$	
Domain	D_{area}	$= 2 \times PPE_{\text{area}} + P \times PE_{\text{area}}$	
Store buffer	SB_{area}	$= 2.464\text{mm}^2$	
L1 cache	$L1_{\text{area}}$	$= 0.363\text{mm}^2/\text{KB}$	
Network switch	N_{area}	$= 0.349\text{mm}^2$	
Cluster	C_{area}	$= D \times D_{\text{area}} + SB_{\text{area}} + L1 \times L1_{\text{area}} + N_{\text{area}}$	
L2 area	$L2_{\text{area}}$	$= 11.78\text{mm}^2/\text{MB}$	
Utilization factor	U	$= 0.94$	
Total WaveScalar processor area	WC_{area}	$= \frac{1}{U}(C \times C_{\text{area}}) + L2_{\text{area}}$	

message per cycle, per PE on the intra-domain network, two messages per domain on the inter-domain network, and two messages per cycle on the network links between clusters. Adding more bandwidth does not significantly increase performance (less than 5%), and using a narrower inter-cluster and inter-domain interconnects reduces the size of a cluster by less than 1%, because the additional wires lay on top of logic that is already present.

The area model contains parameters that enumerate the range of possible WaveScalar processor designs. For parameters D (domains per cluster), P (processors per domain), V (instructions per PE) and M (matching-table entries), we set the range to match constraints imposed by the RTL model. As discussed in Section 7.1, increasing any of these parameters past the maximum value impacts cycle time. The minimum values for M and V reflect restrictions on minimum memory array sizes in our synthesis toolchain.

The ranges in the table allow for over twenty-one thousand WaveScalar processor configurations, but many of them are clearly poor, unbalanced designs, while others are extremely large (up to 12,000mm^2). We can reduce the number of designs dramatically if we apply some simple rules.

First, we bound the die size at 400mm^2 in 90nm to allow for aggressively large yet feasible designs. Next, we remove designs that are clearly inefficient. For instance, it makes no sense to have more than one domain if the design contains fewer than eight PEs per domain. In this case, it is always better to combine the PEs into a single domain, since reducing the domain size does not reduce the cycle time (which is set by the PE's EXECUTE pipeline stage) but does increase communication latency. Similarly, if there are fewer than four domains in the design, there should be only one cluster. Applying these rules and a few more like them reduces the number of designs to 201.

We evaluated all 201 design points on our benchmark set. For the Splash2 applications, we ran each application with a range of thread counts on each design and report

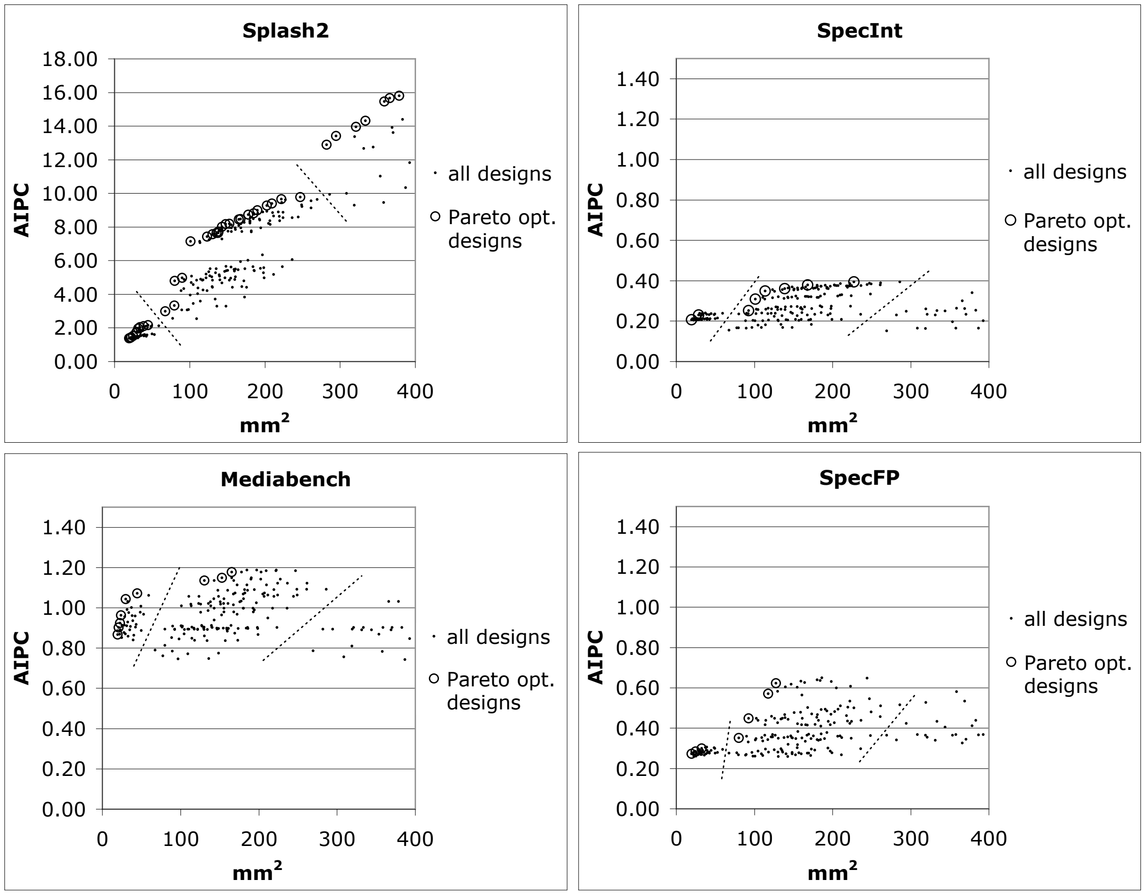


Figure 7.1: **Pareto-optimal WaveScalar designs:** The dotted lines separate designs according the number of clusters they contain. Note the difference in AIPC scale for the Splash2 data.

results for the best-performing thread count. Figure 7.1 shows the results for each group of applications. Each point in the graph represents a configuration, and the circled points are pareto-optimal configurations (i.e., there are no configurations that are smaller *and* achieve better performance). We discuss single- and multi-threaded applications separately and then compare WaveScalar to two other architectures in terms of area efficiency.

7.2.2 Single-threaded workloads

On single-threaded programs WaveScalar suffers from the same problem that single-threaded von Neumann processors face – there is generally very little parallelism available in single-threaded programs written in imperative programming languages. As a result, WaveScalar achieves performance comparable to a single-threaded von Neumann machine. The WaveScalar is, however, much more area efficient than comparable von Neumann designs.

Pareto analysis

The data for all three groups of single-threaded workloads follow the same trend (see Figure 7.1). The configurations fall into three clumps (dotted lines) depending on how many clusters they utilize. Both SpecINT and SpecFP see the most benefit from four cluster designs. On a single cluster they achieve only 58% and 46% (respectively) of the peak performance they attain on four clusters. Mediabench sees smaller gains (9%) from four clusters.

None of the workloads productively utilize a 16 cluster design. In fact, performance decreases for these configurations because instructions become more spread out, increasing communication costs (see below).

Comparison to other single-threaded architectures

To evaluate WaveScalar's single-threaded performance, we compare three different architectures: two WaveCaches and an out-of-order processor. For the out-of-order (OOO) measurements, we use `sim-alpha` configured to model the Alpha EV7 [23, 33], but with the same L1, L2, and main memory latencies we model for WaveScalar.

Figure 7.2 compares all three architectures on the single-threaded benchmarks using AIPC. Of the two WaveCache designs, WS1x1 has better performance on two floating point applications (*ammp* and *equake*). A single cluster is sufficient to hold

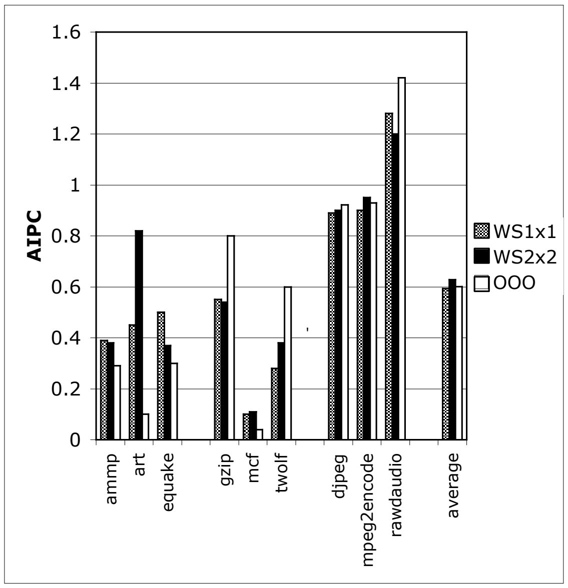


Figure 7.2: **Single-threaded WaveCache vs. superscalar:** On average, both WaveCaches perform comparably to the superscalar.

the working set of instructions for these applications, so moving to a 4-cluster array spreads the instructions out and increases communication costs. The costs take two forms. First, the WC2x2 contains four L1 data caches that must be kept coherent, while WC1x1 contains a single cache, so it can avoid this overhead. Second, the average latency of messages between instructions increases by 20% on average, because some messages must traverse the inter-cluster network. The other applications, except *twolf* and *art*, have very similar performance on both configurations. *Twolf* and *art* have large enough working sets to utilize the additional instruction capacity (*twolf*) or the additional memory bandwidth provided by the four L1 data caches (*art*).

The performance of the WS1x1 compared to OOO does not show a clear winner in terms of raw performance. WS1x1 does better for four applications, outperforming OOO by 4.5× on *art*, 66% on *equake*, 34% on *ammp*, and 2.5× on *mcf*. All these applications are memory-bound (OOO with a perfect memory system performs between 3.6-32× better). Two factors contribute to WaveScalar's superior performance.

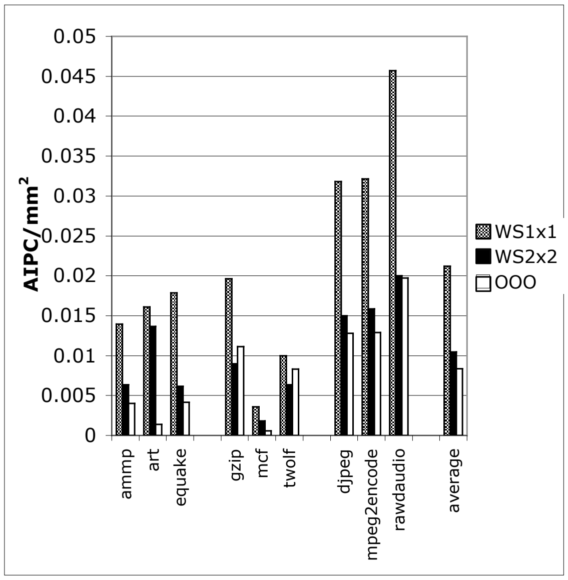


Figure 7.3: **Performance per unit area:** The 1x1 WaveCache is the clear winner in terms of performance per area.

First, WaveScalar's dataflow execution model allows several loop iterations to execute simultaneously. Second, since wave-ordered memory allows many waves to be executing simultaneously, load and store requests can arrive at the store buffer long before they are actually applied to memory. The store buffer can then prefetch the cache lines that the requests will access, so when the requests emerge from the store buffer in the correct order, the data they need is waiting for them.

WaveScalar does less well on integer computations due to frequent function calls. A function can only occur at the end of a wave, because called functions immediately create a new wave. As a result, frequent function calls in the integer applications reduce the size of the waves the compiler can create by 50% on average compared to floating point applications, consequently reducing memory parallelism. *Twolf* and *gzip* are hit hardest by this effect, and OOO outperform WS1x1 by 54% and 32%, respectively. For the rest of the applications, WS1x1 is no more than 10% slower than OOO.

The performance differences between the two architectures are clearer if we take into account the die area required for each processor. To estimate the size of OOO, we examined a die photo of the EV7 in 180nm technology [33, 36]. The entire die is 396mm^2. From this, we subtracted the area devoted to several components that our RTL model does not include (e.g., the IO pads, and inter-chip network controller), but would be present in a real WaveCache. We estimate the remaining area to be $\sim$291mm^2, with $\sim$207mm^2 devoted to 2MB of L2 cache. Scaling all these measurements to 90nm technology yields $\sim$72mm^2 total and 51mm^2 of L2. Measurements from our RTL model show that WC1x1 occupies 48mm^2 (12mm^2 of L2 cache) and WC2x2 occupies 247mm^2 (48mm^2 of L2 cache) in 90nm.

Figure 7.3 shows the area-efficiency of the WaveCaches measured in AIPC/mm^2 compared to OOO. For our single threaded workloads, OOO achieves an area efficiency of 0.008 AIPC/mm^2 and a bottom-line performance of 0.6 AIPC. Depending on the configuration, WaveScalar's efficiency for the single-threaded applications is between 0.004 AIPC/mm^2 (0.71 AIPC; configuration 31) and 0.033 AIPC/mm^2 (0.45 AIPC; configuration 1). WaveScalar configuration 4 closely matches the OOO's performance, but is 50% more area-efficient.

In summary, for most of our workloads, the WaveCache's bottom-line single-threaded AIPC is as good as or better than conventional superscalar designs, and it achieves this level of performance with a less complicated design and in a smaller area.

7.2.3 Multi-threaded workloads

The multi-threaded Splash2 benchmarks provide ample parallelism and can take advantage of much larger WaveScalar designs. They also afford an opportunity to compare WaveScalar's multithreading abilities to those of CMPs.

Pareto analysis

Figure 7.1 (upper left) shows WaveScalar's performance for multithreaded workloads. Because the workloads are multithreaded, they can take advantage of additional area very effectively, as both performance and area increase at nearly the same rate. The dotted lines divide the designs into groups of 1-, 4-, and 16-cluster configurations. The wide, horizontal gap at $\sim$6 IPC is a result of running different numbers of threads. For each configuration, we run a range of different thread counts for each application and report the best overall performance. Designs above the gap generally have larger *virtualization degrees* (i.e., the number of instructions that can reside at a PE) and L1 data caches, allowing them to support more threads simultaneously providing more parallelism.

Table 7.4 lists the pareto-optimal configurations for Splash2 divided into five groups of designs with similar performance (column "Avg. AIPC").

Group A is the single-cluster configurations. Designs 1, 2, 5, and 6 contain only two domains per cluster. Designs 5 and 6 outperform smaller, four-domain designs (3 and 4) because pressure in the L1 data cache outweighs the benefits of additional processing elements.

Moving to four clusters without an L2 cache (Group B) increases the chip's size by 52% and its performance by 38%. Doubling the size of the L1 provides an additional 11% performance boost.

Group C (configurations 13 and 14) "trades-in" large L1 caches for an on-chip L2. For configuration 13, this results in a 43.5% improvement in performance for a negligible increase in area.

The first four 4-cluster designs (Groups B and C) all have much smaller virtualization degrees than the best-performing single-cluster designs (64 vs. 128-256). Configuration 15, the first in Group D, has 128 instructions per PE. The change leads to 43% more performance and requires only 12% additional area.

Table 7.4: **Pareto optimal configurations:** Optimal configurations for the Splash2 applications.

Group	Id	Clusters	Domains/ Cluster	PEs/ Domain	Virt.	Matching Entries	*L*1 (KB)	*L*2 (MB)	Inst. Cap.	Area (mm^2)	Avg. AIPC	AIPC/ mm^2	Area Increase	AIPC Increase
A	1	1	2	8	256	16	8	0	4096	19	1.4	0.07	na	na
	2	1	2	8	256	32	8	0	4096	21	1.4	0.07	6.3%	0.7%
	3	1	4	8	128	16	8	0	4096	22	1.5	0.07	7.4%	4.0%
	4	1	4	8	128	32	8	0	4096	25	1.5	0.06	11.1%	5.0%
	5	1	2	8	256	16	32	0	4096	29	1.7	0.06	16.6%	13.2%
	6	1	2	8	256	32	32	0	4096	30	1.8	0.06	4.3%	1.8%
	7	1	4	8	128	16	32	0	4096	31	2.0	0.06	5.1%	11.9%
	8	1	4	8	128	32	32	0	4096	34	2.0	0.06	7.8%	3.7%
	9	1	4	8	128	64	32	0	4096	39	2.1	0.05	14.5%	3.2%
	10	1	4	8	256	32	32	0	8192	44	2.2	0.05	14.5%	3.1%
B	11	4	4	8	64	16	8	0	8192	67	3.0	0.04	51.9%	37.8%
	12	4	4	8	64	16	16	0	8192	80	3.3	0.04	18.4%	11.5%
C	13	4	4	8	64	16	8	1	8192	80	4.8	0.06	0.2%	43.5%
	14	4	4	8	64	32	8	1	8192	90	5.0	0.06	12.3%	4.0%
D	15	4	4	8	128	16	8	1	16384	101	7.2	0.07	12.5%	43.4%
	16	4	4	8	128	32	16	1	16384	123	7.4	0.06	22.0%	3.9%
	17	4	4	8	128	64	8	1	16384	130	7.6	0.06	5.9%	2.0%
	18	4	4	8	128	32	16	2	16384	136	7.6	0.06	4.1%	0.9%
	19	4	4	8	128	16	32	1	16384	138	7.7	0.06	1.8%	0.1%
	20	4	4	8	128	16	8	4	16384	138	7.7	0.06	0.4%	1.0%
	21	4	4	8	128	64	8	2	16384	143	8.0	0.06	3.1%	3.6%
	22	4	4	8	128	32	32	1	16384	148	8.2	0.06	3.5%	2.1%
	23	4	4	8	256	32	8	1	32768	153	8.2	0.05	3.4%	0.3%
	24	4	4	8	256	32	8	2	32768	165	8.5	0.05	8.2%	3.2%
	25	4	4	8	128	64	32	1	16384	167	8.5	0.05	1.3%	0.2%
	26	4	4	8	256	32	16	2	32768	178	8.7	0.05	6.1%	3.0%
	27	4	4	8	256	64	8	2	32768	185	8.8	0.05	4.1%	1.0%
	28	4	4	8	256	32	32	1	32768	190	9.0	0.05	2.7%	2.0%
	29	4	4	8	256	32	32	2	32768	202	9.3	0.05	6.6%	3.2%
	30	4	4	8	256	64	32	1	32768	209	9.4	0.04	3.5%	1.1%
	31	4	4	8	256	64	32	2	32768	222	9.7	0.04	6.0%	2.9%
	32	4	4	8	256	64	32	4	32768	247	9.8	0.04	11.3%	1.2%
E	33	16	4	8	64	16	8	1	32768	282	12.9	0.05	14.1%	31.8%
	34	16	4	8	64	16	8	2	32768	294	13.4	0.05	4.4%	4.0%
	35	16	4	8	64	32	8	1	32768	321	14.0	0.04	9.1%	4.1%
	36	16	4	8	64	32	8	2	32768	334	14.3	0.04	3.9%	2.6%
	37	16	4	8	64	32	8	4	32768	359	15.5	0.04	7.5%	8.0%
	38	16	4	8	128	16	8	1	65536	366	15.7	0.04	2.0%	1.3%
	39	16	4	8	128	16	8	2	65536	378	15.8	0.04	3.4%	0.9%

The remaining designs in Group D increase slowly in area and performance. Virtualization degree is most important to performance followed by matching table size and cache capacity. With few exceptions, the gains in performance are smaller than the increase in area that they require. As a result, the area efficiency of the designs falls from 0.07 AIPC/mm^2 to 0.04 AIPC/mm^2.

The designs in Group D are the best example of a phenomenon that also occurs in Group A and among the largest pareto-optimal designs for SpecINT and mediabench. In each case, increases in area lead to small changes in performance. This suggests that *within those size ranges* (19-44mm^2 for Group A and 101-247mm^2 for Group D) the designs are as balanced as possible without moving to significantly larger configurations. However, Groups A and D end when the number of PEs increases, demonstrating that the availability of PEs is the key constraint on performance of the smaller designs. For the largest SpecINT and mediabench designs, the data tell a different story. In those cases, the limiting factor is parallelism in the application, and no amount of additional hardware can improve performance.

The configurations in Group E (33-39) have 16 clusters. The transition from 4 to 16 clusters mimics the transition from 1 to 4: The smallest 16 cluster designs have less virtualization and increase their performance by increasing matching table capacity and cache sizes. The larger designs "trade-in" matching table capacity for increased virtualization.

The results demonstrate that WaveScalar is scalable both in terms of area efficiency and peak performance. Design 3 is the most area efficient design, and design 15 uses the same cluster configuration to achieve the same level of efficiency (0.07 AIPC/mm^2). As a result, the designs are approximately a factor of 5 apart in both area and performance. Scaling the same cluster configuration to a 16-cluster machine (configuration 38) reduces the area efficiency to 0.04 AIPC/mm^2, however. A more area-efficient alternative (0.05 AIPC/mm^2) is configuration 33, which corresponds to scaling up configuration 13 and using a slightly smaller L2 data cache.

Design 10 outperforms all the other 1 cluster designs (2.2 AIPC), and configuration 28 uses the same cluster design to achieve 4.2× the performance (9 AIPC) in 4.6× the area. This level of performance is within 5% of the best performing 4-cluster design (configuration 32), demonstrating scalability of raw performance.

This slightly imperfect scalability (the last 5%), however, raises complex questions for chip designers. For instance, if an implementation of configuration 38's cluster is available (e.g., from an implementation of configuration 3), is it more economic to quickly build that larger, slightly less efficient design or to expend the design and verification effort to implement configuration 33 instead? At the very least, it suggests tiled architectures are not perfectly scalable, but rather scalability claims must be carefully analyzed.

Larger designs

To evaluate the WaveCache's multithreaded performance more thoroughly, we simulate a 64-cluster design, representing an aggressive "big iron" processor built in next-generation process technology and evaluated its performance using Splash2. Then, to place the multithreaded results in context with contemporary designs, we compare a smaller, 16-cluster array that could be built today with a range of multithreaded von Neumann processors from the literature. For the workloads the studies have in common, the 16-cluster WaveCache outperforms the von Neumann designs by a factor of between 2 and 16.

We simulate an 8x8 array of clusters (based on configuration 14 in Table 7.4) to model an aggressive, future-generation design. Using the results from our area model scaled to 45nm, we estimate that the processor occupies ∼360mm^2, with an on-chip 16MB L2.

Our performance metric is execution-time speedup relative to a single thread executing on the same WaveCache. We also compare the WaveScalar speedups to those calculated by other researchers for other threaded architectures. Component metrics

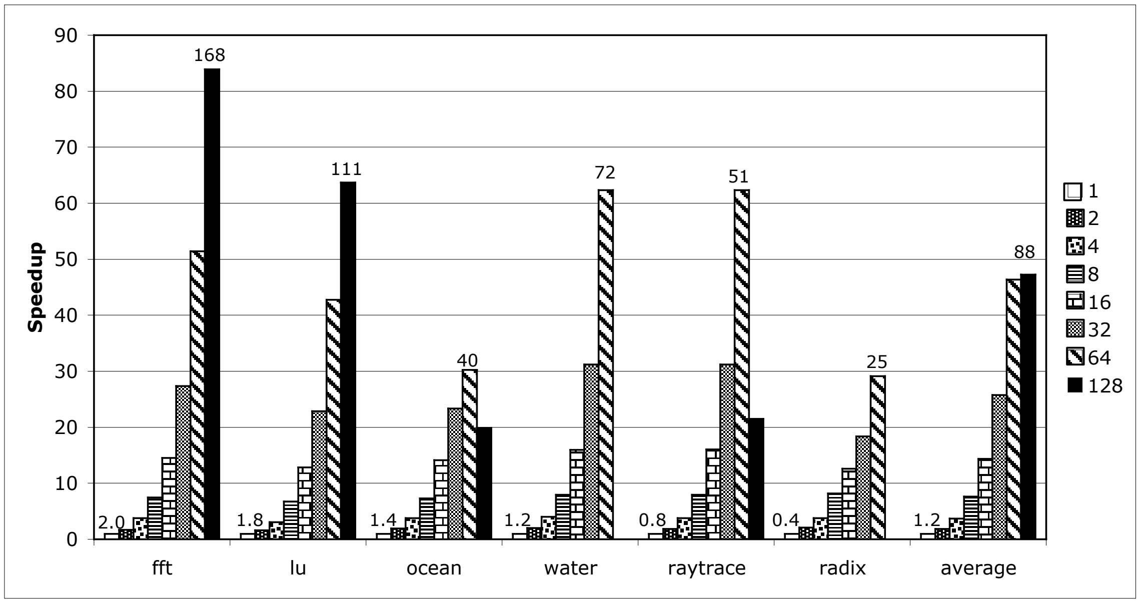


Figure 7.4: **Splash-2 on the WaveCache:** We evaluate each of our Splash-2 benchmarks on the baseline WaveCache with between 1 and 128 threads.

help explain these bottom-line results, where appropriate.

Figure 7.4 contains speedups of multithreaded WaveCaches for all six benchmarks, as compared to their single-threaded running time. The bars represent speedup in total execution time. The numbers above the single-threaded bars are IPC for that configuration. Two benchmarks, *water* and *radix*, cannot utilize 128 threads with the input data set we use, so that value is absent. On average, the WaveCache achieves near-linear speedup (25×) for up to 32 threads. Average performance increases sub-linearly with 128 threads, up to 47× speedup with an average IPC of 88.

Interestingly, increasing beyond 64 threads for *ocean* and *raytrace* reduces performance. The drop-off occurs because of WaveCache congestion from the larger instruction working sets and L1 data evictions due to capacity misses. For example, going from 64 to 128 threads, *ocean* suffers 18% more WaveCache instruction misses than would be expected from the additional compulsory misses. In addition, the operand matching cache miss rate increases by 23%. Finally, the data cache miss rate, which

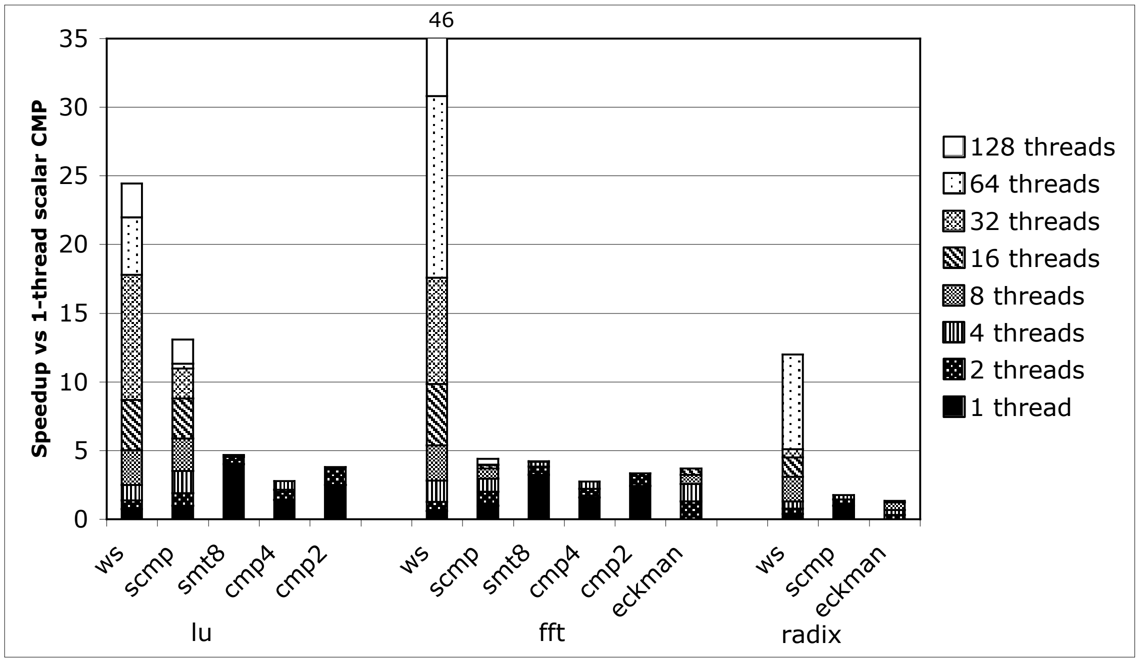


Figure 7.5: **Performance comparison of various architectures:** WaveScalar's multi-threaded performance compared to several other multi-threaded architectures.

is essentially constant for up to 32 threads, doubles as the number of threads scales to 128. This additional pressure on the memory system increases *ocean*'s memory access latency by a factor of eleven.

The same factors that cause the performance of *ocean* and *raytrace* to suffer when the number of threads exceeds 64 also reduce the rate of speedup improvement for other applications as the number of threads increases. For example, the WaveCache instruction miss rate quadruples for *lu* when the number of threads increases from 64 to 128, curbing speedup. In contrast, FFT, with its relatively small per-thread working set of instructions and data, does not tax these resources, and so achieves better speedup with up to 128 threads.

Comparison to other multi-threaded architectures

To understand how WaveCache performance compares with other architectures we perform two sets of experiments. In the first, we compare WaveCache performance to other multi-threaded architectures on three Splash-2 kernels: *lu*, *fft* and *radix*. Then we compare its area efficiency to the Niagara processor from Sun.

To compare our Splash2 performance to other threaded processors, we present results from several sources in addition to our own WaveCache simulator. For CMP configurations we performed our own experiments using a simple in-order core (*scmp*), as well as measurements from [40] and [24]. Comparing data from such diverse sources is difficult, and drawing precise conclusions about the results is not possible. However, we believe that the measurements are valuable for the broad trends they reveal.

To make the comparison as equitable as possible, we use a smaller, 4x4 WaveCache for these studies. Our RTL model gives an area of 253mm^2 for this design (we assume an off-chip, 16 MB L2 cache and increase its access time from 10 to 20 cycles). While we do not have precise area measurements for the other architectures, the most aggressive configurations (i.e., most cores or functional units) are in the same ball park with respect to size.

To facilitate the comparison of performance numbers of these different sources, we normalized all performance numbers to the performance of a simulated scalar processor with a 5-stage pipeline. The processor has 16KB data and instruction caches, and a 1MB L2 cache, all 4-way set associative. The L2 hit latency is 12 cycles, and the memory access latency of 200 cycles matches that of the WaveCache.

Figure 7.5 shows the results. The stacked bars represent the increase in performance contributed by executing with more threads. The bars labeled *ws* depict the performance of the WaveCache. The bars labeled *scmp* represent the performance of a CMP whose cores are the scalar processors described above with 1MB of L2 cache per processor core. These processors are connected via a shared bus between private L1 caches and a shared L2 cache. Memory is sequentially consistent and uses a snooping version of the protocol WaveScalar's L1 data caches use. Up to 4 accesses to the shared memory may overlap. For the CMPs the stacked bars represent increased performance from simulating more processor cores. The 4- and 8-core bars loosely model *Hydra* [31] and a single *Piranha* chip [10], respectively.

The bars labeled *smt8*, *cmp4* and *cmp2* are the 8-threaded SMT and 4- and 2-core out-of-order CMPs from [40]. We extracted their running times from data provided by the authors. Memory latency is low on these systems (dozens of cycles) compared to expected future latencies, and all configurations share the L1 data- and instruction caches.

To compare the results from [24] (labeled *ekman* in the figure), which are normalized to the performance of their 2-core CMP, we simulated a superscalar with a configuration similar to one of these cores and halved the reported execution time; we then used this figure as an estimate of absolute baseline performance. In [24], the authors fixed the execution resources for all configurations, and partitioned them among an increasing number of decreasingly wide CMP cores. For example, the 2-thread component of the *ekman* bars is the performance of a 2-core CMP in which each core has a fetch width of 8, while the 16-thread component represents the performance of 16 cores with a fetch-width of 1. Latency to main memory is 384 cycles, and latency to the L2 cache is 12 cycles.

The data show that the WaveCache can handily outperform the other architectures at high thread counts. It executes 1.8× to 10.9× faster than *scmp*, 5.2× to 10.8× faster than *smt8*, and 6.4× to 16.6× faster than the various out-of-order CMP configurations. Component metrics show that the WaveCache's performance benefits arise from its use of point-to-point communication, rather than a system-wide broadcast mechanism, and from the latency-tolerance of its dataflow execution model. The former enables scaling to large numbers of clusters and threads, while the latter helps mask the increased memory latency incurred by the directory protocol and the high load-use penalty on the L1 data cache.

The performance of all systems eventually plateaus when some bottleneck resource saturates. For *scmp* this resource is shared L2 bus bandwidth. Bus saturation occurs at 16 processors for LU, 8 for FFT and 2 for RADIX[1]. For the other von Neumann CMP systems, the fixed allocation of execution resources is the limit [40], resulting in a decrease in per-processor IPC. For example, in *ekman*, per-processor IPC drops 50% as the number of processors increases from 4 to 16 for RADIX and FFT. On the WaveCache, speedup plateaus when the working set of all the threads equals its instruction capacity. This offers the WaveCache the opportunity to tune the number of threads to the amount of on-chip resources. With their static partitioning of execution resources across processors, this option is absent for CMPs; and the monolithic nature of SMT architectures prevents scaling to large numbers of thread contexts.

Finally, we compare the WaveCache to Sun's Niagara processor [37] in terms of area efficiency. Niagara is an 8-way CMP targeted at large multi-threaded workloads. We are unaware of any published evaluation of Splash2 on Niagara, but [37] shows Niagara running at 4.4 IPC (0.55 IPC/core) at 1.2 GHz (an approximately 10% faster clock than WaveScalar's). Niagara's die measures 379mm^2, giving an efficiency of 0.01 IPC/mm^2. Since each Niagara core is single-issue and in-order, and all eight cores share a floating point unit, the theoretical maximum efficiency is 0.02 IPC/mm^2. The least efficient pareto optimal WaveScalar design (configuration 39) is twice as efficient (0.04 AIPC/mm^2).

Discussion

The WaveCache has clear promise as a multiprocessing platform. In the 90nm technology available today, we could easily build a WaveCache that would outperform a range

[1]While a 128-core *scmp* with a more sophisticated coherence system might perform more competitively with the WaveCache on RADIX and FFT, studies of these systems are not present in the literature, and it is not clear what their optimal memory system design would be.

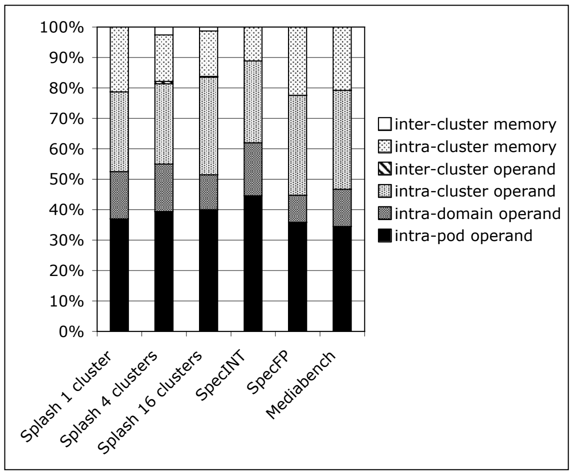


Figure 7.6: **The distribution of traffic in the WaveScalar processor:** The vast majority of traffic in the WaveScalar processor is confined within a single cluster and, for many applications, over half travels only over the intra-domain interconnect.

of CMP designs both in terms of bottom-line performance and area efficiency. Scaling multi-threaded WaveScalar systems beyond a single die is also feasible. WaveScalar's execution model makes and requires no guarantees about communication latency, so using several WaveCache processors to construct a larger computing substrate is a possibility.

7.3 Network traffic

One goal of WaveScalar's hierarchical interconnect is to isolate as much traffic as possible in the lower levels of the hierarchy, namely, within a PE, a pod or a domain. Figure 7.6 breaks down all network traffic according to these levels. It reveals the extent to which the hierarchy succeeds on all three workloads, and for the parallel applications, on a variety of WaveScalar processor sizes. On average 40% of network traffic travels from a PE to itself or to the other PE in its pod, and 52% of traffic remains within a domain. For multi-cluster configurations, on average just 1.5% of traffic traverses the inter-cluster interconnect. The graph also distinguishes between operand data and memory/coherence traffic. Operand data accounts for the vast majority of messages, 80% on average, with memory traffic less than 20%.

These results demonstrate the scalability of communication performance on the WaveScalar processor. Applications that require only a small patch of the processor, such as Spec2000, can execute without ever paying the price for long distance communication. In addition, the distribution of traffic types barely changes with the number of clusters, indicating that the interconnect partitioning scheme is scalable. Message latency does increase with the number of clusters (by 12% from 1 to 16 clusters), but as we mentioned above, overall performance still scales linearly. One reason for the scalability is that the WaveScalar instruction placement algorithms isolate individual Splash threads into different portions of the die. Consequently, although the average distance between two clusters in the processor increases from 0 (since there is only one cluster) to 2.8, the average distance that a message travels increases by only 6%. For the same reason, network congestion increases by only 4%.

7.4 Discussion

We have presented a design space analysis for WaveScalar processors implemented in an ASIC tool flow. However, our conclusions also apply to full-custom WaveScalar designs and, in some ways, to other tiled architectures such as CMPs.

The two main conclusions would not change in a full-custom design. For the first conclusion, that WaveScalar is inherently scalable both in terms of area efficiency and raw performance, our data provide a lower bound on WaveScalar's ability to scale. Custom implementations should lead to smaller designs, faster designs, or both. Our second conclusion, that WaveScalar's area efficiency compares favorably to more conventional designs, would hold in a full custom design for the same reason.

WaveScalar's hierarchical interconnect would serve well as the interconnect for an aggressive CMP design, and our results concerning the scalability and performance of the network would apply to that domain as well. In a conventional CMP design, the network carries coherence traffic, but as researchers strive to make CMPs easier to program it might make sense to support other types of data, such as MPI messages, as well. In that case, our data demonstrate the value of localizing communication as much as possible and the feasibility of using a single network for both coherence traffic and data messaging.

Chapter 8

WAVESCALAR'S DATAFLOW FEATURES

The WaveScalar instruction set and hardware design we have described so far replicates the functionality of a von Neumann processor or a CMP composed of von Neumann processors. Providing these capabilities is essential for WaveScalar to be a viable alternative to von Neumann architectures, but it is not the limit of what WaveScalar can do.

This chapter exploits WaveScalar's dataflow underpinning to achieve two things that conventional von Neumann machines cannot. First, it provides a second, *unordered* memory interface that is similar to the token-passing interface in Section 3.4.1. The unordered interface is built to express memory parallelism. It bypasses the wave-ordered store buffer and accesses the L1 cache directly, avoiding the overhead of the wave-ordering hardware. Because the unordered operations do not go through the store buffer, they can arrive at the L1 cache in any order or in parallel.

Second, the WaveCache can support very fine-grain threads. On von Neumann machines the amount of hardware devoted to a thread is fixed (e.g., one core on a CMP or one thread context on an SMT machine), and the number of threads that can execute at once is relatively small. On the WaveCache, the number of physical store buffers limits the number of threads that use wave-ordered memory simultaneously, but any number of threads can use the unordered interface at one time. In addition, spawning these threads is even less expensive than spawning threads that use the wave-ordered interface (Section 5.1), so it is feasible to break a program up into 100s of parallel, fine-grain threads.

We begin by describing the unordered memory interface. Then we use it and fine-grain threads to express large amounts of parallelism in three application kernels. Finally, we combine the two styles of programming to parallelize *equake* from the Spec2000 floating point suite, and demonstrate that by combining WaveScalar's ability to run both kinds (i.e., coarse-grain, von Neumann-style threads and fine-grain, dataflow-style threads), we can achieve performance greater than utilizing either alone, in this case, a 9× speedup.

8.1 Unordered memory

As described, WaveScalar's only mechanism for accessing memory is the wave-ordered memory interface. The interface is necessary for executing conventional programs, but it can only express limited parallelism (i.e., by using ripple numbers). WaveScalar's unordered interface makes a different trade-off: it cannot efficiently provide the sequential ordering that conventional programs require, but it excels at expressing parallelism, because it eliminates unneeded ordering constraints and avoids contention for the store buffer. Because of this, it allows programmers or compilers to express and exploit memory parallelism when they know it exists.

Like all other dataflow instructions, unordered operations are only constrained by their static data dependences. This means that if two unordered memory operations are not directly or indirectly data-dependent, they can execute in any order. Programmers and compilers can exploit this fact to express parallelism between memory operations that can safely execute out of order; however, they need a mechanism to enforce ordering among those that cannot.

To illustrate, consider a STORE and a LOAD that could potentially access the same address. If, for correct execution, the LOAD must see the value written by the STORE (i.e., a read-after-write dependence), then the thread must ensure that the LOAD does not execute until the STORE has finished. If the thread uses wave-ordered memory, the store buffer enforces this constraint; however, since unordered memory operations bypass the wave-ordered interface, unordered accesses must use a different mechanism.

To ensure that the Load executes after the Store, there must be a data dependence between them. This means memory operations must produce an output token that can be passed to the operations that follow. Loads already do this, because they return a value from memory. We modify Stores to produce a value when they complete. The value that the token carries is unimportant, since its only purpose is to signal that the Store is complete. In our implementation it is always zero. We call unordered Loads and Stores, Load-Unordered and Store-Unordered-Ack, respectively.

8.1.1 Performance evaluation

To demonstrate the potential of unordered memory, we implemented three traditionally parallel but memory-intensive kernels – matrix multiply (MMUL), longest common subsequence (LCS), and a finite input response filter (FIR) – in three different styles and compared their performance. *Serial coarse-grain* uses a single thread written in C. *Parallel coarse-grain* is a coarse-grain parallelized version, also written in C, that uses the coarse-grain threading mechanisms described in Chapter 5. *Unordered* uses a single coarse-grain thread written in C to control a pool of fine-grain threads that use unordered memory, written in WaveScalar assembly. We call these *unordered threads*.

For each application, we tuned the number of threads and the array tile size to achieve the best performance possible for a particular implementation. MMUL multiplies 128×128 entry matrices, LCS compares strings of 1024 characters, and FIR filters 8192 inputs with 256 taps. They use between 32 (*FIR*) and 1000 (*LCS*) threads. Each version is run to completion.

Figure 8.1 depicts the performance of each algorithm executing on the 8x8 WaveCache described in Section 7.2.3. On the left, it shows speedup over the serial implementation, and, on the right, average units of work completed per cycle. For MMUL

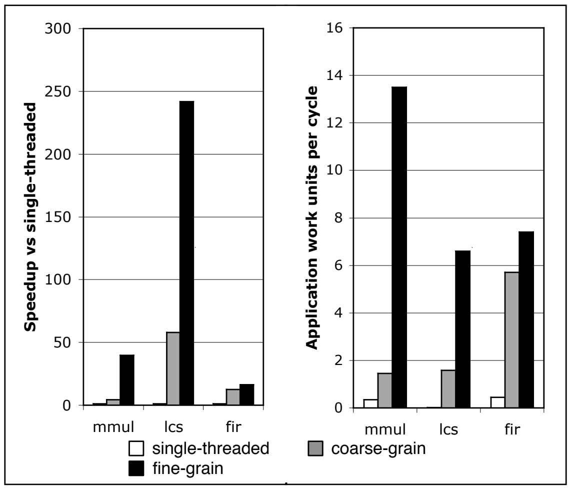


Figure 8.1: **Fine-grain performance:** Performance of three different implementation styles

and FIR, the unit of work selected is a multiply-accumulate, while for LCS, it is a character comparison. We use application-specific performance metrics, because they are more informative than IPC when comparing the three implementations. For all three kernels, the unordered implementations achieve superior performance because they exploit more parallelism.

The benefits stem from two sources. First, the unordered implementations can use more threads. It would be easy to write a pthread-based version that spawned 100s or 1000s of threads, but the WaveCache cannot execute that many ordered threads at once, since there are not enough store buffers. Secondly, the unordered threads' memory operations can execute in parallel. As a result, the fine-grain, unordered implementation exploit more inter- and intra-thread parallelism. *MMUL* is the best example; it executes 27 memory operations per cycle on average (about one per every two clusters), compared to just 6 for the coarse-grain version.

FIR and *LCS* are less memory-bound than *MMUL* because they load values (input samples for *FIR* and characters for *LCS*) from memory once and then pass them from thread to thread directly. For these two applications the limiting factor is inter-cluster network bandwidth. Both algorithms involve a great deal of inter-thread communication, and since the computation uses the entire 8x8 array of clusters, inter-cluster communication is unavoidable. For *LCS* 27% of messages travel across the inter-cluster network compared to 0.4-1% for the single-threaded and coarse-grain versions, and the messages move 3.6 times more slowly due to congestion. *FIR* displays similar behavior. A different placement algorithm might alleviate much of this problem and improve performance further by placing the instructions for communicating threads near one another.

8.2 Mixing threading models

In Chapter 5, we explained the extensions to WaveScalar that support coarse-grain, pthread-style threads. In the previous section, we introduced two lightweight memory instructions that enable fine-grain threads and unordered memory. In this section, we combine these two models; the result is a hybrid programming model that enables coarse- and fine-grain threads to coexist in the same application. We begin with two examples that illustrate how ordered and unordered memory operations can be used together. Then, we exploit all of our threading techniques to improve the performance of Spec2000's *equake* by a factor of nine.

8.2.1 Mixing ordered and unordered memory

A key strength of our ordered and unordered memory mechanisms is their ability to coexist in the same application. Sections of an application that have independent and easily analyzable memory access patterns (e.g., matrix manipulations and stream processing) can use the unordered interface, while difficult to analyze portions (e.g., pointer-chasing codes) can use wave-ordered memory.

We describe two ways to combine ordered and unordered memory accesses. The first turns off wave-ordered memory, uses the unordered interface, and then reinstates wave-ordering. The second, more flexible approach allows the ordered and unordered

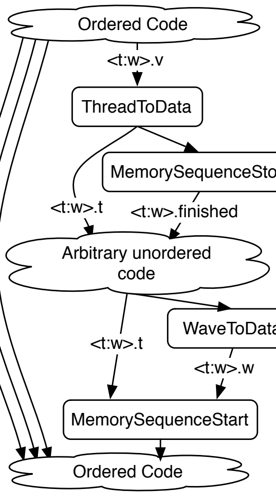


Figure 8.2: **Transitioning between memory interfaces:** The transition from ordered to unordered memory and back again.

interfaces to exist simultaneously.

Example 1 Figure 8.2 shows a code sequence that transitions from wave-ordered memory to unordered memory and back again. The process is quite similar to terminating and restarting a pthread-style thread. At the end of the ordered code, a THREAD-TO-DATA instruction extracts the current THREAD-ID, and a MEMORY-SEQUENCE-STOP instruction terminates the current memory ordering. MEMORY-SEQUENCE-STOP outputs a value, labeled *finished* in the figure, after all preceding wave-ordered memory operations have completed. The *finished* token triggers the dependent, unordered memory operations, ensuring that they do not execute until the earlier, ordered-memory accesses have completed.

After the unordered portion has executed, a MEMORY-SEQUENCE-START creates a new, ordered memory sequence using the THREAD-ID extracted previously. In principle, the new thread need not have the same THREAD-ID as the original ordered thread. In practice, however, this is convenient, as it allows values to flow directly from the first ordered section to the second (the curved arcs on the left side of the

```
struct {
    int x,y;
} point;

foo(point *v, int *p, int *q)
{
    point t;
    *p = 0;
    t.x = v->x;
    t.y = v->y;
    return *q;
}
```

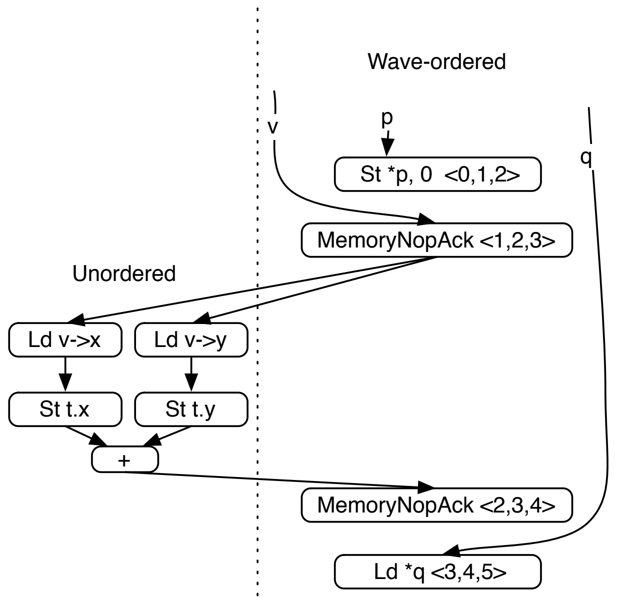


Figure 8.3: **Using ordered and unordered memory together:** A simple example where MEMORY-NOP-ACK is used to combine ordered and unordered memory operations to express memory parallelism.

figure) without THREAD-ID manipulation instructions.

Example 2: In many cases, a compiler may be unable to determine the targets of some memory operations. The wave-ordered memory interface must remain intact to handle these hard-to-analyze accesses. Meanwhile, unordered memory accesses from analyzable operations can simply bypass the wave-ordering interface. This approach allows the two memory interfaces to coexist in the same thread.

Figure 8.3 shows how the MEMORY-NOP-ACK instruction from Section 5.2.1 allows programs to take advantage of this technique. Recall that MEMORY-NOP-ACK is a wave-ordered memory operation that operates like a memory fence instruction, returning a value when it completes. We use it here to synchronize ordered and unordered memory accesses. In function `foo`, the loads and stores that copy `*v` into `t` can execute in parallel but must wait for the store to `*p`, which could point to any address. Likewise, the load from address `*q` cannot proceed until the copy is complete. The wave-ordered memory system guarantees that the store to `*p`, the two Memory-Nop-Acks, and the load from `*q` fire in the order shown (top to bottom). The data dependences between the first Memory-Nop-Ack and the unordered loads at left ensure that the copy occurs after the first store. The Add instruction simply coalesces the outputs from the two Store-Unordered-Ack instructions into a trigger for the second Memory-Nop-Ack that ensures the copy is complete before the final load.

8.2.2 A detailed example: equake

To demonstrate that mixing the two threading styles is not only possible but also profitable, we optimized *equake* from the SPEC2000 benchmark suite. *equake* spends most of its time in the function *smvp*, with the bulk of the remainder confined to a single loop in the program's *main* function. In the discussion below, we refer to this loop in *main* as *sim*.

We exploit both ordered, coarse-grain and unordered, fine-grain threads in *equake*. The key loops in *sim* are data independent, so we parallelized them using coarse-grain threads that process a work queue of blocks of iterations. This optimization improves *sim*'s performance by 2.2×, but speeds up *equake* by only a factor of 1.6 overall.

Next, we used the unordered memory interface to exploit fine-grain parallelism in *smvp*. Two opportunities present themselves. First, each iteration of *smvp*'s nested loops loads data from several arrays. Since these arrays are read-only, we used unordered loads to bypass wave-ordered memory, allowing loads from several iterations to execute in parallel. Second, we targeted a set of irregular cross-iteration dependences in *smvp*'s inner loop that are caused by updating an array of sums. These cross-iteration dependences make it difficult to profitably coarse-grain-parallelize the loop. However, the Thread-Coordinate instruction lets us extract fine-grain parallelism despite these dependences, since it efficiently passes array elements from PE to PE and guarantees that only one thread can hold a particular value at a time. This idiom is inspired by M-structures [11], a dataflow-style memory element. Rewriting *smvp* with unordered memory and THREAD-COORDINATE improves that functions performance by 11×, but overall gains are only 2.2×.

When both coarse-grain and fine-grain threading are used together, *equake* speeds up by a factor of 9.0. This result demonstrates that coarse-grain, pthread-style threads and fine-grain, unordered threads can be combined to accelerate a single application beyond what either technique can achieve in isolation.

Chapter 9
RELATED WORK

WaveScalar is the most recent in a long line of dataflow processor architectures, but its goal differs in some ways from its predecessors. Previous dataflow machines focused (mostly) on performance for scientific computing, and the key advantage of dataflow was the parallelism it could express and exploit. WaveScalar's approach differs in two respects. First, it targets general purpose programs and imperative programming languages. Second, although it takes advantage of dataflow parallelism, it exploits dataflow's decentralized execution model to address the challenges that face modern chip designers.

In this regard, WaveScalar resembles other tiled architectures that researchers have proposed. These designs strive to reduce design effort by replicating hardware structures across the silicon die. The dataflow execution model is a natural match for such a design.

This chapter places WaveScalar in context relative to both previous dataflow designs and modern tiled architectures. It also addresses several arguments against the dataflow model as the basis for scalable processor designs.

9.1 Previous dataflow designs

Dataflow execution models, by definition, share two characteristics. First, they represent programs as dataflow graphs. The nodes in the graph are operations and the dataflow arcs between them carry data values from one operation to another. Second, execution occurs according to the dataflow firing rule. That is, an operation may execute when its inputs are available.

A wide range of execution models fall within this definition, and researchers have designed and built an equally wide range of hardware designs to implement them. We categorize dataflow research efforts along four axes. First, dataflow models vary in how they deal with dynamic instances of individual values and when operations become eligible to execute. Second, different models provide different granularities of dataflow operations. Some models restrict the nodes in the dataflow graph to single instructions while others allow for long-running threads. Third, models provide varying types of mutable memories, usually in the form of implicitly synchronized memories. The fourth axis is the organization of the hardware implementation and, in particular, how the hardware implements the dataflow firing rule. We discuss each of these axes in turn. Table 9.1 summarizes the discussion.

9.1.1 Execution model

Dennis [21] described the first dataflow execution model with a corresponding processor architecture. His initial design was the *static dataflow* model. Static dataflow systems allow a single data value to be present on an arc in the dataflow graph at one time and use the simplest version of the dataflow firing rule. In static dataflow systems an operation can fire when a value is present on each input arc. A dataflow graph that produces multiple simultaneous values on a dataflow arc is malformed.

Static dataflow's simplicity limits its generality and the amount of parallelism it can express. For example, they cannot support general recursion, because recursive dataflow graphs can easily result in multiple, unconsumed values waiting on dataflow arcs. Also, multiple iterations of a loop cannot execute simultaneously.

These disadvantages limited static dataflow's success as a processor execution model, but the idea has found applications in other domains. For instance, electrical circuits are essentially static dataflow graphs with charge and current acting as values that electrical components compute on.

The alternative to static dataflow is *dynamic dataflow*. Dynamic dataflow ma-

Table 9.1: **Dataflow architectures:** Summary of dataflow architecture characteristics.

Architecture	Year	Execution model	Operation granularity	Tag matching method	Special memory structures
Dennis [21]	1975	static dataflow	fine	static frame-based	none
DDM1 [20]	1978	dynamic fifo[a]	fine	FIFO-based	none
TTDA [8]	1980	dynamic	fine	CAMs	I-structures
Manchester [30]	1985	dynamic	fine	hashing	none
Sigma-1 [56]	1986	dynamic	fine	CAMs	none
Epsilon [28]	1989	static	fine	static frame-based	none
EM-4 [52, 54]	1989	dynamic	coarse	frame-based	thread-local data
P-Risc [47]	1989	dynamic	coarse	frame-based	I-structures
Epsilon 2 [29]	1990	dynamic	coarse	frame-based	I-structures, lists, arrays
Monsoon [50]	1990	dynamic	fine/coarse	frame-based	I-structures
*T [48]	1992	dynamic	coarse	hardware support for thread synchronization	I-structures, thread-local data
WaveScalar [59, 60]	2003	dynamic	fine	hashing	wave-ordered memory

[a]Dataflow arcs can hold multiple untagged values. They are consumed in FIFO order.

chines allow multiple values on an arc at once, allowing for both recursion and parallel execution of loop iterations. The cost, however, comes in a more complicated dataflow firing rule. Instead of waiting for *any* inputs to be available, operations must wait until a set of matching inputs are available.

Dynamic dataflow machines differentiate between instances of values by adding a *tag* to each value. The combination of a data value and a tag is called a *token*. In dynamic dataflow machines an operation can only execute if a set of tokens with the same tag is present on its inputs. Normal operations copy the tag from the input values to the outputs, but the instruction sets often provide special tag management instructions to modify tags (i.e., WaveScalar's DATA-TO-WAVE and WAVE-TO-DATA instructions).

The MIT Tagged Token Dataflow Architecture (TTDA) [8] was one of the earliest dynamic dataflow machines, and nearly all subsequent dataflow machines, including WaveScalar, use the dynamic dataflow model. WaveScalar differs from previous dynamic systems in providing a two-part tag (wave number and thread id) and giving the programmer complete control over the tags. Previous machines provided special hardware or firmware for automatic tag management.

Tag management increases the burden on the programmer slightly but allows more flexible use of the available space of tags. For instance, the fine-grain threading mechanism described in Section 8.1.1 relies upon creative use of wave number and thread ids.

9.1.2 Operation granularity

In the abstract, the dataflow graph of an application defines a partial order over a set of operations. Different dataflow models and their corresponding hardware implementations support different granularities of operation. Designs such as TTDA [8], the Manchester machine [30], SIGMA-1 [56], Epsilon [28], Monsoon [50], and WaveScalar used individual instructions as operations. Other designs, such as the EM4 [54, 52], P-Risc [47], *T [48], and the Threaded Abstract Machine (TAM) [16] allowed for coarse-grain operations (often called threads) and provided hardware acceleration for synchronization operations.

The shift from fine-grained dataflow to coarse-grain dataflow was a response to the high cost of frequent synchronization in fine-grain dataflow machines. Coarse-grain operations make synchronization less frequent. They reduce communication overhead, because each operation runs to completion on one processing element, confining communication within an operation to a single processor.

Despite the strong trend in previous dataflow research toward coarse-grain operations, WaveScalar uses fine-grain operations. The dataflow languages that older dataflow machines targeted provide parallelism at a wide range of granularities. Designers could adjust the granularity of their operations to balance synchronization and communication costs in the technology available at the time.

WaveScalar does not have this luxury, because one of its goals is to efficiently execute programs written in C. These programs tend to have a small amount of instruction-level parallel (ILP) and, if they are multi-threaded, some very coarse-grain, pthread-style parallelism. There is little or no parallelism in between. As a result, WaveScalar *must* use fine-grain operations to exploit fine-grain ILP, so it uses different techniques to address the costs of synchronization and communication, namely careful instruction placement, a hierarchical interconnect, and a highly-optimized tag matching scheme. WaveScalar's hierarchical interconnect provides fast, local communication and our instruction placement scheme keeps most communication localized, reducing communication costs. In addition, we apply a technique called k-loop bounding [15] to reduce overflow and keep matching cheap.

9.1.3 Memory

Designers of the earliest dataflow machines intended them to run purely functional languages that had no notion of mutable state. However, support for mutable state can make programming some applications much easier. Many dataflow languages incorporate some notion of mutable state along with special-purpose hardware to make it fast.

Researchers have studied three different dataflow memory systems. We discuss I-structures and M-structures below. Section 3.4.1 discusses the third approach, token-passing.

I-structures Functional languages initialize variables when they are declared and disallow modifying their values. This eliminates the possibility of read-after-write data hazards: The variable always contains the correct value so any read is guaranteed to see it. Dataflow language designers recognized that this approach restricts parallelism, because an array must be completely initialized before its elements can be accessed. Ideally, one thread could fill in the array while another thread accessed the initialized elements.

Dataflow languages such Id [49] and SISAL [25] provide this ability with I-structures [9]. I-structures are write-once memory structures. When a program allocates an I-structure it is *empty* and contains no value. A program can write, or fill in, an I-structure at most once. Reading from an empty I-structure blocks until the I-structure is full. Reading from a full I-structure returns the value it holds. In the array example above, one thread allocates an array of I-structures and starts filling them in. The second thread can attempt to read entries of the array but will block if it tries to access an empty I-structure.

M-structures M-structures [11] provide check-in/check-out semantics for variables. Reading from a full M-structure removes the value, and a write fills the value back in. Attempting to read from an empty M-structure blocks until the value is returned.

A typical example of M-structures in action is a histogram. Each bucket is an M-structure, and a group of threads add elements to the buckets concurrently. Since addition is commutative, the order of the updates is irrelevant, but they must be sequentialized. M-structures provide precisely the necessary semantics.

Although I- and M-structures provide mutable state, they do not provide the ordering semantics of imperative languages. Since WaveScalar must support these languages, it provides wave-ordered memory. However, WaveScalar can emulate M-structures using THREAD-COORDINATE.

9.1.4 Dataflow hardware

All dataflow machines follow a similar high-level design: A set of processing elements (PEs) connected by some sort of interconnect. The individual PEs implement the dataflow firing rule and execute instructions. Some designs also provides special I- or M-structure processors to implement those constructs.

The individual PEs are typically pipelined. Tokens arrive at the top of the pipeline and flow into a *matching unit* that checks whether the token completes a set of inputs to a instruction. If it does, the instruction issues to the functional unit. Eventually, the result of the operation leaves the pipeline and travels over the interconnect network to its destination PE.

The most significant difference among dataflow machines is how the matching units implements the dataflow firing rule. Both static and dynamic machines must address this issue because a processing element usually handles more than one instruction. Static machines must check each value to see if it completes the set of inputs for an instruction (i.e., if the destination instruction identifier matches the identifier of a previously generated value). For dynamic machines the matching unit must check both the destination instruction identifier and the tag. For fine-grained dataflow architectures efficient tag matching is crucial, but for coarse-grain designs it is less frequent and, therefore, less important.

There are two main challenges. First, since the number of unconsumed input tokens is potentially unbounded, the matching unit at each PE should be fast and accommodate as many unconsumed input tokens as possible to minimize overflow. Second, when overflow does occur, the architecture must provide a mechanism to handle additional tokens. Research reports do not provide detailed descriptions of overflow mechanisms, but the hope is that overflow is an infrequent occurrence.

Fine-grain dataflow architectures have used three techniques for tag matching: associative search, hashing, frame-based lookup. We discuss each in turn.

Associative search The earliest dynamic dataflow machines [8] used associative search. Special-purpose content-addressable memories (CAMs) check the tag of an incoming token against all waiting tokens. CAMs are complex and require a great deal of hardware to implement. This technique quickly fell from favor.

Hashing Architectures such as the Manchester machine [30] used hashing to replace expensive CAMs with conventional memories. As tokens arrive, the hardware computes a hash of its tag and destination instruction identifier and uses the hash to index into a memory. If the word of memory contains a matching token, a match occurs and the instruction issues. If the word of memory contains a token with a different tag, the hardware invokes an overflow mechanism. Finally, if the word of memory is empty, the hardware places the new token there to await its matching token. WaveScalar uses a hashing scheme. Section 4.2 describes it in detail.

Frame-based lookup Frame-based lookup, used in Monsoon [50] and EM-4 [54], among others, relies on the compiler to eliminate the possibility of token overflow. The compiler breaks the program into *blocks* (e.g., basic blocks, hyperblocks, or functions). When the hardware needs to execute a block, it first allocates a *frame* of memory. The frame contains one entry for each two-input instruction in the block (one-input instructions do not need an entry because they do not require tag matching), and each word of the frame carries a *presence bit* that denotes whether the entry is full or empty. When an instruction generates a value, the hardware reads the frame entry for the destination instruction. If the entry is full, a match occurs. Otherwise, the hardware writes the new value into the frame entry to await its mate.

Static dataflow machines can be seen as frame-based machines in which a single frame covers the entire program and only one frame is ever allocated. Dennis's first dataflow proposal [21] follows this approach.

9.2 Tiled architectures

The WaveCache hardware design in Chapters 4 and 5 is a tiled architecture. Broadly speaking, a tiled architecture is a processor design that uses an array of basic building blocks of silicon to construct a larger processor.

Tiled architectures provide three advantages over traditional monolithic designs. First, they reduce design complexity by emphasizing design reuse. WaveScalar exploits this principle at several levels (PE, domain, and cluster). Second, tiled designs seek to avoid long wires. In modern technology, wire delay dominates the cost of computation. Wires in most tiled architectures span no more than a single tile, ensuring that wire length does not increase with the number of tiles. Finally, tiled architectures seek to be scalable. An ideal tiled architecture would scale to any number of tiles both in terms of functional correctness and in terms of performance.

Several research groups have proposed tiled architectures with widely varying tile designs. Smart Memories [42] provides multiple types of tiles (e.g., processing elements and reconfigurable memory elements). This approach allows greater freedom in configuring an entire processor, since the mix of tiles can vary from one instantiation to the next, perhaps avoiding the difficulties in naive scaling that we found in our study.

The TRIPS [46, 53] processor uses dataflow ideas to build a hybrid von Neumann/dataflow machine. The TRIPS processor uses a program counter to guide execution, but instead of moving from one instruction to the next, the TRIPS PC selects *frames* (similar to hyperblocks [41]) of instructions for execution in an array of 16 processing elements that make up a TRIPS processor.

Despite high-level similarities between waves and frames and the WaveScalar and TRIPS PE designs, the two architectures are quite different. In TRIPS, a register file at the top of the array holds values that pass from one frame to another. Each TRIPS PE can hold multiple instructions, so each PE requires multiple input buffers. However, execution follows the static dataflow model, making tag matching logic unnecessary.

Using dataflow execution within a von Neumann processor is the same approach taken by out-of-order superscalars, but the TRIPS design avoids the long wires and broadcast structures that make conventional out-of-order processors non-scalable. However, because it uses a program counter to select blocks of instructions for execution, TRIPS must speculate aggressively. Mapping a frame of instructions onto the PE array takes several cycles, so the TRIPS processor speculatively maps frames onto the PEs ahead of time. WaveScalar does not suffer from this problem because its dynamic dataflow execution model allows instructions to remain in the grid for many executions, obviating the need for speculation. The disadvantage of WaveScalar's approach is the need for complex tag-matching hardware to support dynamic dataflow execution

The two projects also have much in common. Both take a hybrid static/dynamic approach to scheduling instruction execution by carefully placing instructions in an array of processing elements and then allowing execution to proceed dynamically. This places both architectures between extremely dynamic out-of-order superscalar designs and fully statically scheduled VLIW machines. Those designs have run into problems, because fully dynamic scheduling does not scale and static scheduling can be very difficult in practice. A hybrid approach will be necessary, but it is unclear whether either WaveScalar or TRIPS strikes the optimal balance.

WaveScalar and TRIPS also take similar approaches to ordering memory operations. TRIPS uses load/store IDs (LSIDs) [57] to order memory operations within a single frame. Like the sequence numbers in wave-ordered memory, LSIDs provide ordering among the memory operations. However, the TRIPS scheme provides no mechanism for detecting whether a memory operation will actually execute during a specific dynamic execution of a frame. Instead, TRIPS guarantees that memory operations that access the same address will execute in the correct order and modify the consistency model to treat frames of instructions as atomic operations. LSID-based memory ordering requires memory disambiguation hardware that increases the complexity of the design relative to WaveScalar's wave-ordering store buffer.

The RAW project [61] uses a simple processor core as a tile and builds a tightly-coupled multiprocessor. The RAW processor provides for several different execution models. The compiler can statically schedule a single program to run across all the tiles, effectively turning RAW into a VLIW-style processor. Alternatively, the cores can run threads from a larger computation that communicate using RAW's tightly-integrate, inter-processor, message-passing mechanism. The former approach is reminiscent of the coarse-grain dataflow processing described in Section 9.1.2.

The RAW project presents one vision for what an advanced CMP might look like, and a careful study of its area-efficiency would be worthwhile. One such study of the RAW architecture [44] shares similar goals with ours (Section 7.2), but it takes a purely analytical approach and creates models for both processor configurations and applications. That study was primarily concerned with finding the optimal configuration for a particular application and problem size, rather than a more universally scalable, general purpose design.

9.3 Objections to dataflow

As Table 9.1 demonstrates, dataflow processors fell out of favor between the early 1990's and the early twenty-first century. Several factors contributed to this, including cheaper, higher-performance commodity processors and the integration of dataflow ideas into out-of-order processors. But there was also a feeling that dataflow was flawed as an execution model, at least given the technology available at the time.

Culler [17] articulated the difficulties with dataflow as two key problems. First, the memory hierarchy limits the amount of latency a dataflow machine can hide. A processor can only hide latency (and keep busy) by executing instructions whose inputs are at the top level of the memory hierarchy. If the top of the memory hierarchy is too small, the processor will sit idle, waiting for the inputs to arrive.

Second, the dataflow firing rule is naive, since it ignores locality in execution. Ideally, a dataflow machine would execute program fragments that are related to one another to exploit locality and prevent "thrashing" at the top of the memory hierarchy.

Culler's first argument does not apply to WaveScalar because of changes in technology. The relatively small number of transistors available on a single die in the early 1990's meant that execution resources were expensive. As a result hiding latency and keeping the processors busy were the keys to performance. Things have changed significantly. WaveScalar sidesteps the need to keep many data values "near" a handful of processing elements by building many processing elements, each with a small amount of nearby memory for storing unmatched tokens.

WaveScalar's carefully-tuned placement system addresses the problems with locality and scheduling. As described in Section 4.6 the placement algorithm attempts to schedule dependent instructions on a single PE, ensuring that a large fraction (30-40%) of message traffic occurs within a single pod. This reduces the need for buffering a large number of tokens because, tokens tend to be consumed shortly after they are produced. In addition, k-loop bounding reduces the frequency of token overflow, further improving locality.

Chapter 10

CONCLUSIONS AND FUTURE WORK

The WaveScalar instruction set, execution model, and hardware architecture demonstrate that dataflow computing is a viable alternative to the von Neumann model. WaveScalar overcomes the limitations of previous dataflow architectures, addresses many of the challenges that modern processors face, exploits many types of parallelism, and provides high performance on a wide range of applications.

WaveScalar makes two key contributions to dataflow instruction sets. The first is wave-ordered memory. By adding a simple set of annotations, WaveScalar provides the sequential memory semantics that imperative programming languages require without destroying the parallelism that the dataflow model expresses. Without wave-ordered memory, or a scheme like it, dataflow processing has little chance of being a viable competitor to von Neumann processors.

Work on wave-ordered memory is not complete, and two key areas remain to be explored. The evaluation in Chapter 3 uses our binary translator-based toolchain, and the binary translator cannot make full use of wave-ordered memory. In particular, it does not have access to good aliasing information or high-level structural information about the program. Better aliasing knowledge would allow for more aggressive use of the ripple annotation (and the unordered memory interface, see below), and structural information would allow the compiler to generate larger, more efficient waves. Work in these directions is underway.

More broadly, wave-ordered memory demonstrates the value of encoding program structure explicitly in the instruction set. This idea has applications beyond WaveScalar or even dataflow processing. Extending conventional instruction sets to encode more information about program behavior, control structure, or memory usage is one approach to improving performance in the face of stringent technological constraints. The challenge is finding concise, flexible ways to express useful information to the processor.

WaveScalar's second contribution to dataflow instruction sets is its ability to combine the imperative memory semantics that wave-ordered memory provides with the massive amounts of parallelism that a dataflow execution model can express. WaveScalar's unordered memory interface provides the means to exploit this type of parallelism among memory operations, while providing well-defined semantics for combining ordered and unordered accesses in the same program.

In addition to intermingling accesses via the ordered and unordered memory interfaces, WaveScalar also allows the programmer to combine coarse- and fine-grain threading styles to express a range of different types of parallelism. The combination of two threading models and two memory interfaces provides WaveScalar programmers with enormous flexibility in expressing parallelism. This leads to better performance, but it also has the potential to ease the burden on programmers, because they can express the types of parallelism that are most natural in the program at hand. Fully realizing this goal will require additional work, however. Currently the only way to exploit the fine-grain threading and unordered memory is via hand-written dataflow assembly (a difficult undertaking). Ongoing compiler work aims to apply both interfaces to single-threaded codes automatically. Other approaches include porting a compiler for a dataflow programming language to WaveScalar or extending an imperative language to support these interfaces explicitly.

The WaveScalar hardware architecture in Chapters 4 and 5 is designed to be scalable and of low complexity, and to effectively exploit the different types of parallelism the WaveScalar ISA can express. Our WaveScalar design reduces design complexity by combining identical components hierarchically. This approach reduces the amount of hardware that must be designed and verified. It also eases the impact of manufacturing defects. If a portion of the chip, for instance a PE, is manufactured incorrectly, that PE can be disabled without significantly impacting the rest of the chip. Investigating what kinds of manufacturing errors the WaveScalar architecture can tolerate and how it would adapt to them is the subject of future work.

To evaluate WaveScalar we used a combination of RTL synthesis (for area and delay measurements) and cycle-level simulation (for performance measurements). Our results are encouraging. An exhaustive search of WaveScalar's design space reveals that the WaveScalar processor is scalable. For chip sizes from 39mm^2 to 294mm^2, it scales almost perfectly, realizing a 6.4-fold increase in performance for a 7.5$\times$ increase in chip area. We also found that the most area-efficient single-cluster design scales to the most efficient 4- and 16-cluster designs.

Eventually, we plan to extend this to include power in addition to area and performance. Synthesis tools can extract power consumption estimates from our RTL model, and by combining these with event counts from our simulator, we can compute WaveScalar's power requirements. Using this infrastructure, we can evaluate different approaches to power management in a WaveScalar processor, including clock- and power-gating at the PE, domain, and cluster levels as well as power-aware instruction placement. Including power will also force us to carefully consider aspects of the design like the interconnect that have little impact on area, but might contribute significantly to the power budget.

We will also extend the study to include a wider range of applications. Currently, the limitations of our toolchain and runtime environment prevent us from running large-scale "commercial" workloads that would put significantly more pressure on the memory system and the WaveScalar's instruction capacity than our current suite of applications. Running these workloads on WaveScalar will lead to further refinements in the architecture. For instance, our current cache coherence protocol contains only simple optimizations for migratory sharing, because preliminary limit studies showed that for our applications the benefits of more complicated schemes were small. For applications with threads that span several clusters and have more inter-thread communication, a more sophisticated scheme might perform better.

Using the scalable design points from our pareto analysis, we compared WaveScalar to more conventional architectures. Our results show that a well-tuned WaveScalar design outperforms more conventional CMP designs on multi-threaded workloads by between 2× and 11× depending on the workload, and it achieves single-threaded performance similar to that of an aggressive out-of-order superscalar but in 30% less area.

The results for WaveScalar's fine-grain threading and unordered memory interfaces are even more encouraging. For kernels, these interfaces can provide speedups of between 16× and 240×. When both memory interfaces and both threading styles are brought to bear (something that is impossible in a conventional CMP), they increased overall performance by 9× for one application.

WaveScalar demonstrates that dataflow computing is a viable alternative to conventional von Neumann processing. More important, however, are questions of order, parallelism, distributed control, and scalability that this work has explored. These problems are at the heart of one of the central challenges facing computer system designers today: How can we design and build computer systems that are both easy to implement and will deliver performance that scales with technology? This thesis used WaveScalar to begin addressing this question.

BIBLIOGRAPHY

[1] Cadence website. http://www.cadence.com.

[2] Synopsys website. http://www.synopsys.com.

[3] TCB013GHP - TSMC 0.13um core library. Taiwan Semiconductor Manufacturing Company, Ltd. Release 1.0. October 2003.

[4] S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12), December 1996.

[5] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[6] J. M. Arnold, D. A. Buell, and E. G. Davis. Splash 2. In *Proceedings of the 4th Symposium on Parallel Algorithms and Architectures*, 1992.

[7] Arvind. Dataflow: Passing the token. 32nd Annual International Symposium on Computer Architecture Keynote, June 2005.

[8] Arvind and R.S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1990.

[9] Arvind, R.S. Nikhil, and K. K. Pingali. I-structures: Data structures for parallel computing. *ACM Transaction on Programming Languages and Systems*, 11(4):598–632, 1989.

[10] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[11] P. S. Barth, R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional languages with state. Technical Report MIT/LCS/TR-327, MIT, 1991.

[12] M. Beck, R. Johnson, and K. Pingali. From control flow to data flow. *Journal of Parallel and Distributed Computing*, 12:118–129, 1991.

[13] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. *SIGPLAN Notices*, 39(11):14–26, 2004.

[14] D. Chinnery and K. Keutzer. *Closing the Gap Between ASIC & Custom*. Kluwer Academic Publishers, 2003.

[15] D. E. Culler. *Managing Parallelism and Resources in Scientific Dataflow Programs*. PhD thesis, Massachusetts Institute of Technology, March 1990.

[16] D. E. Culler, A. Sah, K. E. Schauser, T. Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[17] D. E. Culler, K. E. Schauser, and T. von Eicken. Two fundamental limits on dataflow multiprocessing. In *Proceedings of the IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, 1993.

[18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[19] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computing*, 36(5):547–553, 1987.

[20] A. L. Davis. The architecure and system method of DDM1: A recursively structured data driven machine. In *Proceedings of the 5th Annual Symposium on Computer Architecture*, 1978.

[21] J. B. Dennis. A preliminary architecture for a basic dataflow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture*, 1975.

[22] J. B. Dennis. First version data flow procedure language. Technical Report MAC TM61, MIT Laboratory for Computer Science, May 1991.

[23] R. Desikan, D.C. Burger, S.W. Keckler, and T.M. Austin. Sim-alpha: A validated, execution-driven Alpha 21264 simulator. Technical Report TR-01-23, UT-Austin Computer Sciences, 2001.

[24] M. Ekman and P. Stenström. Performance and power impact of issue width in chip-multiprocessor cores. In *Proceedings of the International Conference on Parallel Processing*, 2003.

[25] J. T. Feo, P. J. Miller, and S. K. Skedzielewski. SISAL90. In *Proceedings of High Performance Functional Computing*, 1995.

[26] S. C. Goldstein and M. Budiu. NanoFabrics: Spatial computing using molecular electronics. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[27] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficent synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989.

[28] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes. The Epsilon dataflow processor. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989.

[29] V.G. Grafe and J.E. Hoch. The Epsilon-2 hybrid dataflow architecture. In *Proceedings of the 35th IEEE Computer Society International Conference*, 1990.

[30] J. R Gurd, C. C Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.

[31] L. Hammond, B.A. Hubbert, M. Siu, M.K. Prabhu, M. Chen, and K. Olukolun. The Stanford Hydra CMP. *IEEE Micro*, 20, March/April 2000.

[32] M. S. Hrishikesh, D. Burger, N. P. Jouppi, S. W. Keckler, K. I. Farkas, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[33] A. Jain, W. Anderson, T. Benninghoff, and D. E. Berucci. A 1.2GHz Alpha microprocessor with 44.8GB/s chip pin bandwidth. In *Proceedings of the IEEE International Solid-State Circuits Conference*, 2001.

[34] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. Sing Lee. Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor. In *Proceedings of the 26nd Annual International Symposium on Computer Architecture*, 1998.

[35] M. Kishi, H. Yasuhara, and Y. Kawamura. DDDP-a distributed data driven processor. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, 1983.

[36] K. Krewel. Alpha EV7 processor: A high-performance tradition continues. *Microprocessor Report*, April 2005.

[37] J. Laudon. Performance/watt: the new server focus. *SIGARCH Computer Architecture News*, 33(4):5–13, 2005.

[38] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.

[39] W. Lee, R. Barua, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a RAW machine. In *Proceedings of the 8th Annual International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-VIII*, 1998.

[40] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, 1997.

[41] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, 1992.

[42] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: a modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[43] M. Mercaldi, M. Oskin, and S. E. Eggers. A performance model to guide instruction scheduling on spatial computers. In submission to the Symposium on Parallel Algorithms and Architectures".

[44] C. A. Moritz, D. Yeung, and A. Agarwal. Exploring performance-cost optimal designs for RAW microprocessors. In *Proceedings of the International IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.

[45] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, 2003.

[46] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, 2001.

[47] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989.

[48] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[49] R.S. Nikhil. The parallel programming language Id and its compilation for parallel machines. In *Proceedings of the Workshop on Massive Paralleism: Hardware, Programming and Applications*, 1990.

[50] G. M. Papadopoulos and D. E. Culler. Monsoon: an explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990.

[51] G. M. Papadopoulos and K. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991.

[52] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An architecture of a dataflow single chip processor. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989.

[53] K. Sankaralingam, R. Nagarajan, H.Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[54] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura. Thread-based programming for the EM-4 hybrid dataflow machine. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[55] T. Shimada, K. Hiraki, and K. Nishida. An architecture of a data flow machien and its evaluation. In *COMPCON Digest of Papers*, 1984.

[56] T. Shimada, K. Hiraki, K. Nishida, and S. Sekiguchi. Evaluation of a prototype data flow processor of the SIGMA-1 for scientific computations. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, 1986.

[57] A. Smith, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, K. S. McKinley, and J. Burrill. Compiling for EDGE architectures. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[58] SPEC. SPEC CPU 2000 benchmark specifications, 2000. SPEC2000 Benchmark Release.

[59] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, 2003.

[60] S. Swanson, A. Putnam, K. Michelson, M. Mercaldi, A. Petersen, A. Schwerin, M. Oskin, and S. J. Eggers. Area-performance trade-offs in tiled dataflow architectures. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[61] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. Evaluation of the RAW microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[62] D. Tullsen, J. Lo, S. Eggers, and H. Levy. Supporting fine-grain synchronization on a simultaneous multithreaded processor. In *Proceedings of the 5th Annual International Symposium on High Performance Computer Architecture*, 1999.

VITA

Steven Swanson was born and raised in Pocatello, Idaho, but he currently lives in Seattle. He earned a Bachelor of Science in Computer Science and Mathematics at the University of Puget Sound in Tacoma, Washington in 1999 and a Master of Science in Computer Science and Engineering from the University of Washington in 2001. In 2006, he earned a Doctor of Philosophy at the University of Washington in Computer Science and Engineering.