The WaveScalar Architecture

Steven Swanson Andrew Schwerin Martha Mercaldi
Andrew Petersen Andrew Putnam Ken Michelson
Mark Oskin Susan Eggers
Computer Science & Engineering
University of Washington
{swanson,schwerin,mercaldi,petersen,aputnam,ken,oskin,eggers}@cs.washington.edu

Abstract:

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 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 paper 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 the WaveCache achieves nearly linear speedup with up to 64 threads and can sustain 7-14 multiply-accumulates per cycle on fine-grain threaded versions of well-known kernels. Finally, we apply both styles of threading to equake from spec2000 and speed it up by 9x compared to the serial version.

Keywords: WaveScalar, Dataflow computing, Multithreading

1 Introduction

It is widely accepted that Moore's Law will hold for the next decade. However, although more transistors will be available, simply scaling up current architectures will not convert them into commensurate increases in performance [1]. This resulting gap between the increases in performance we have come to expect and those that larger versions of existing architectures will be able to 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.

We propose a new instruction set architecture (ISA), called WaveScalar [2], that addresses these challenges by building on the dataflow execution model [3]. The dataflow execution model is well-suited to running on a decentralized, scalable processor, because it is inherently decentralized. In this model, instructions execute when their inputs are available, and detecting this condition can be done locally for each instruction. The global

coordination that the von Neumann model relies on, in the form of a program counter, is not required. In addition, the dataflow model allows programmers and compilers to express parallelism explicitly, instead of relying on the underlying hardware (e.g., an out-of-order superscalar) to extract it.

WaveScalar exploits these properties of the dataflow model, and 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. Instead they used special, dataflow languages that limited their usefulness. A recent ISCA keynote address [4] noted that if dataflow systems are to become a viable alternative to the von Neumann status quo, they must enforce sequentiality on memory operations without severely reducing parallelism among other instructions. WaveScalar addresses this challenge with a memory ordering scheme, called *wave-ordered memory*, that efficiently provides the memory ordering that imperative languages need.

Using this memory ordering scheme, WaveScalar supports conventional single-threaded and pthread-style multithreaded applications. It also efficiently supports fine-grain threads that can consist of only a handful of instructions. Programmers can combine these different thread models in the same program or even in the same function. Our data show that applying diverse styles of threading to a single program can expose significant parallelism in code that would otherwise be difficult to fully 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 sea of processing nodes in a substrate that effectively replaces the central processor and instruction cache of a conventional system. The WaveCache loads instructions from memory and assigns them to processing elements for execution. The instructions remain at their processing elements for a large number, potentially millions, of invocations. As the working set of instructions for the application changes, the WaveCache evicts unneeded instructions and loads the necessary ones into vacant processing elements.

This paper describes and evaluates the WaveScalar ISA and WaveCache architecture. First, we describe those aspects of WaveScalar's ISA and the WaveCache architecture that are required for executing single-threaded applications, including the wave-ordered memory interface. We evaluate the performance of a small, simulated WaveCache on several single-threaded applications. Our data demonstrate that this WaveCache performs comparably to a modern out-of-order superscalar design, but requires only $\sim$38% as much silicon area.

Next, we extend WaveScalar and the WaveCache to support conventional pthread-style threading. The changes to WaveScalar include light-weight dataflow synchronization primitives and support for multiple, independent sequences of wave-ordered memory operations. The multithreaded WaveCache achieves nearly linear speedup on the six Splash2 parallel benchmarks that we use.

Finally, we delve into WaveScalar's dataflow underpinnings, the advantages they provide, and how programs can combine them conventional multi-threading. We describe WaveScalar's "unordered" memory interface and show how it combines with fine-grain threading to reveal substantial parallelism. Fully utilizing these techniques requires a custom compiler which is not yet complete, so we evaluate these two features by using hand-coding three common kernels and rewriting part of the *equake* benchmark to use a combination of fine- and coarse-grain threading styles. The results demonstrate that these techniques speed up the kernels by between 16 and 240 times and *equake* by a factor of 9 compared to the serial versions.

The rest of this paper is organized as follows. Sections 2 and 3 describe the single-threaded WaveScalar ISA and WaveCache architecture, respectively. Section 4 then evaluates them. Section 5 describes WaveScalar's coarse-grain threading facilities and the changes to the WaveCache that support them. Section 6 presents WaveScalar's dataflow-based facilities that support fine-grain parallelism and illustrates how one can combine both threading style to enhance performance. Finally, Section 7 concludes.

2 Single-threaded WaveScalar

The dataflow model that WaveScalar uses is fundamentally different than the von Neumann model that dominates conventional designs, but both models accomplish many of the same tasks in order to execute single threaded programs written in conventional programming languages. For example, both must determine

which instructions to execute and provide a facility for conditional execution; they must pass operands from one instruction to one another and they must access memory.

For many of these tasks, WaveScalar borrows from previous dataflow machines. Its interface to memory, however, is unique and is one of its primary contributions to dataflow computing. The WaveScalar memory interface provides an efficient method for encoding memory ordering information in a dataflow model, enabling efficient execution of programs written in imperative programming languages. Most earlier dataflow machines could not efficiently execute codes written in imperative languages, because they could not easily enforce the memory semantics these programs require.

To provide context for our description, we first describe how the von Neumann model accomplishes the tasks outlined above and why the von Neumann model is inherently centralized. Then we describe how WaveScalar's model accomplishes the same goals in a decentralized manner and how WaveScalar's memory interface works. WaveScalar's decentralized execution model provides the basis for the decentralized, general purpose hardware architecture in Section 3.

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 that read the value are data dependent on the writing instruction.

To access memory, programs issue Load and Store instructions. A key tenet of the von Neumann model is the set of memory semantics it provides: that loads and stores occur (or appear to occur) in the order in which the PC fetched them. Enforcing this order is required to preserve read-after-write, write-after-write, and write-after-read dependences between instructions. Modern, imperative languages, such as C, C++, or Java, provide essentially identical memory semantics and rely on the von Neumann architecture's ability to implement those semantics efficiently.

At its heart, the von Neumann model describes execution as a linear, centralized process. A single program counter guides execution and there is always exactly one instruction that, according to the model, should execute next. This is both a strength and a weakness. On one hand, it makes control transfer easy, tightly bounds the amount of state the processor must maintain, and provides a simple set of memory semantics. History has also demonstrated that constructing processors based on the model is feasible (and extremely profitable!). On the other hand, the model expresses no parallelism. While the performance of its processors has improved exponentially for over three decades, continued scalability is uncertain.

2.2 WaveScalar's ISA

The dataflow execution model has no PC to guide instruction fetch and memory ordering and no register file to serve as a conduit of data values between dependent instructions. Instead, it views instructions as nodes in a dataflow graph that only execute after they have received their input values. Memory operations execute in the same data-driven fashion, which may result in their being executed out of the program's linear order. However, although the model provides no total ordering of a program's instructions, it does enforce the partial orders that a program's dataflow graph defines. Since individual partial orders are data-independent, they can be executed in parallel, providing the dataflow model with an inherent means of expressing parallelism of arbitrary granularity. In particular, the granularity of parallelism is determined by the length of a data-dependent path. For all operations, data values are passed directly from producer instructions to consumer instructions without intervening accesses to a register file.

Dataflow's advantages are its explicit expression of parallelism among dataflow paths and its decentralized execution model that obviates the need for a program counter or any other centralized structure to control instruction execution. However, these 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. WaveScalar handles control using the same technique as previous dataflow machines (described in Section 2.2.2), but overcomes the problem of memory access order with a novel architectural technique called

wave-ordered memory [2] (described in Section 2.2.5). Wave-ordered memory essentially creates a "chain" of dependent memory operations at the architectural level; the hardware then guarantees that the operations execute in the order the chain defines.

Below we describe the WaveScalar ISA in detail. Much 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. Readers already familiar with dataflow execution could skim Sections 2.2.1, 2.2.2, and 2.2.4.

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 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 '←'. 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` $+1$ would be the SUBTRACT instruction.

Unlike the PC-driven von Neumann model, execution of the dataflow graph 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 1, once inputs A and B are ready, the ADD can fire and produce the left-hand input to the DIVIDE. Likewise, once C is available, the SUBTRACT computes the other input to the DIVIDE instruction. The DIVIDE then executes and produces D.

The dataflow firing rule is inherently decentralized, because it allows each instruction to act autonomously, waiting for inputs to arrive and generating outputs. Portions of the dataflow graph that are not explicitly data-dependent do not communicate at all.

2.2.2 Control flow

Dataflow's decentralized execution algorithm makes control transfers more difficult to implement. Instead of steering a single PC through the executable, so the processor executes one path instead of the other, WaveScalar steers values into one part of the dataflow graph and prevents them from flowing into another. It can also use predication to perform both computations and later discard the results on the wrong path. 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.

WaveScalar uses STEER instructions to steer values to the correct path and ϕ instructions for predication. The STEER [5] 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(b) shows a simple conditional implemented with STEER instructions. STEER instructions correspond most directly to traditional branch instructions, and they are required for implementing loops. 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.

The ϕ instruction [6] takes two input values and a boolean selector input and, 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 are desirable, because they remove the selector input from the critical path of some computations and therefore increase parallelism. They are also wasteful, however, because they discard the unselected input. Figure 2(c) shows ϕ instructions in action.

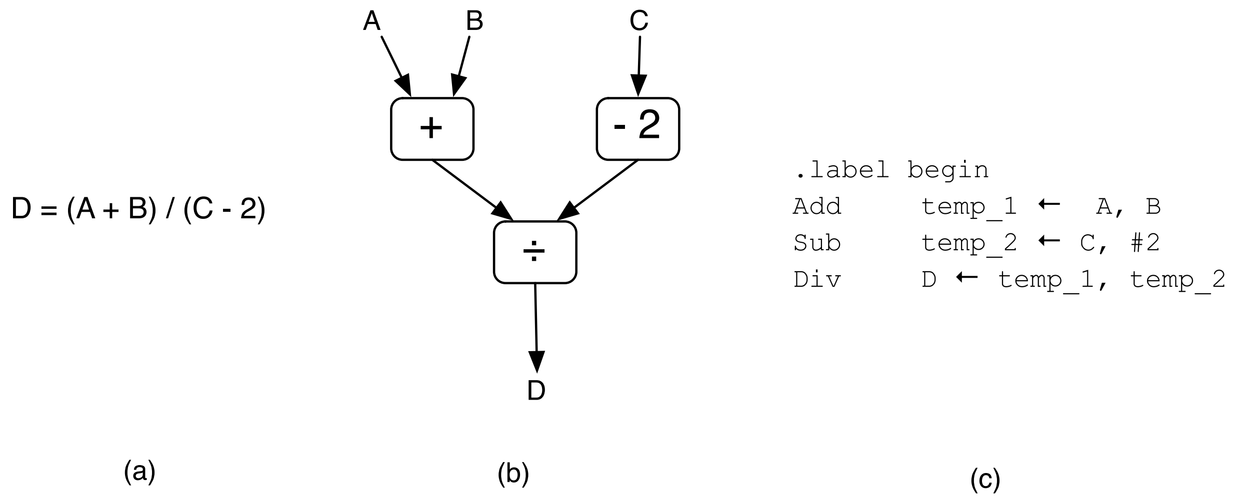

Figure 1: **A Simple dataflow fragment:** A simple program statement (a), its dataflow graph (b), and the corresponding WaveScalar assembly (c). The order of the WaveScalar assembly statements is unimportant, since they will be executed in dataflow fashion.

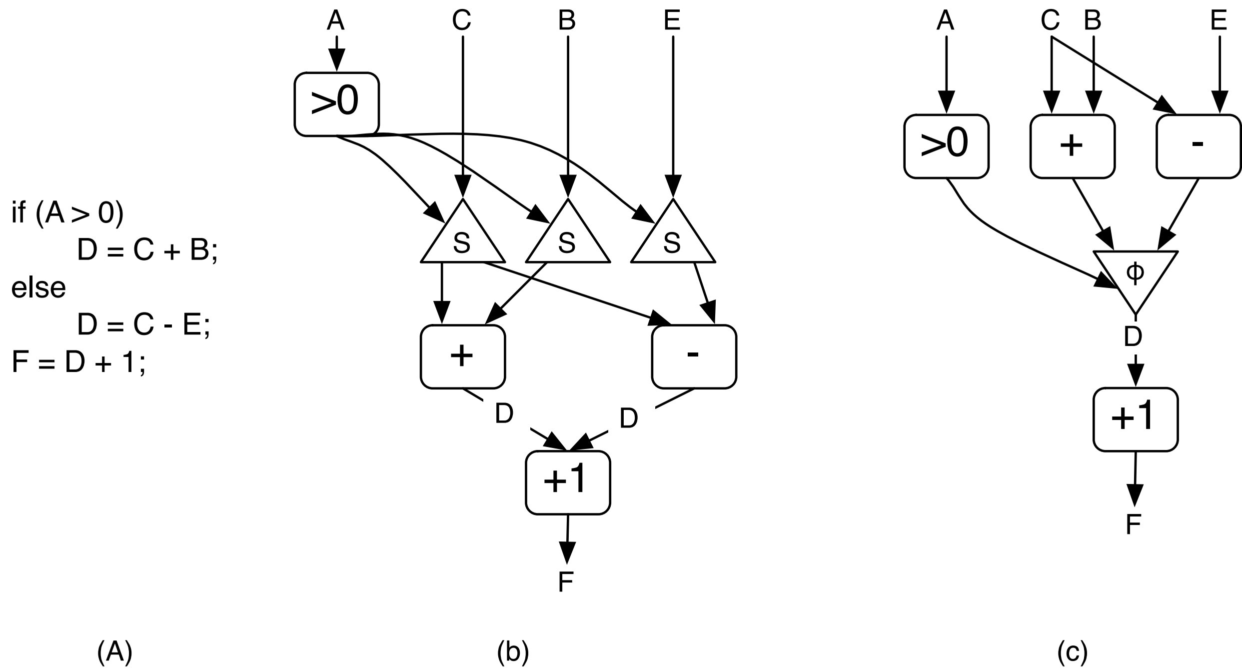

Figure 2: **Implementing control in WaveScalar:** An IF-THEN-ELSE construct (a) and equivalent dataflow representations. In (b) STEER instructions (triangles labeled 's') ensure that only one side of the branch executes, while (c) computes both sides and a ϕ instruction selects the result to use.

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 3(a) shows a simple loop that both 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 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 makes no guarantee about how long it takes a data value to flow along a given dataflow arc. 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 or wise.

Previous dataflow machines provided one of two solutions. In the first, *static dataflow* [3, 7], 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 COMPARE and INCREMENT instructions from producing a new value before the old values had been consumed. While this restriction resolves the ambiguity between different value instances, it also reduces parallelism by preventing multiple iterations of a loop from executing simultaneously and makes recursion difficult to support.

A second model, *dynamic dataflow* [8, 9, 10, 11, 12], tags each data value with an identifier and allows multiple values to wait at the input to an instruction. The dataflow firing rule is modified so that an instruction fires only when tokens with matching tags are available on all its inputs [1]. The combination of a data value and its tag is called a *token*. 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* [2]. We denote a WaveScalar token with wave-number W and value v as $W.v$. Instead of assigning different *wave-numbers* to different instances of specific instructions (as most dynamic dataflow machines did), WaveScalar assigns them to compiler-delineated portions of the dataflow graph called *waves*. Waves are similar to hyperblocks [13], but they are more general, since they can contain control-flow joins and can have more than one entrance. They cannot contain loops. Figure 3(c) shows the example loop divided into waves (as shown by the dotted lines). At the top of each wave is a set of WAVE-ADVANCE instructions (the small diamonds), each of which increments the wave number of the value that passes through it.

Assume the code before the loop is wave number 0. When the code executes, the two CONST instructions will produce 0.0 (wave number 0, value 0). The WAVE-ADVANCE instructions will take these as input and each will output 1.0, which will propagate through the body of the loop as before. At the end of the loop, the right-side 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.

In addition to allowing WaveScalar to extract parallelism, wave-numbers also play a key role in enforcing memory ordering (Section 2.2.5).

[1] The execution model does not specify where the data values are stored or how matching takes place. Efficiently storing and matching input tokens is a key challenge in dynamic dataflow architecture, and Section 3 discusses it.

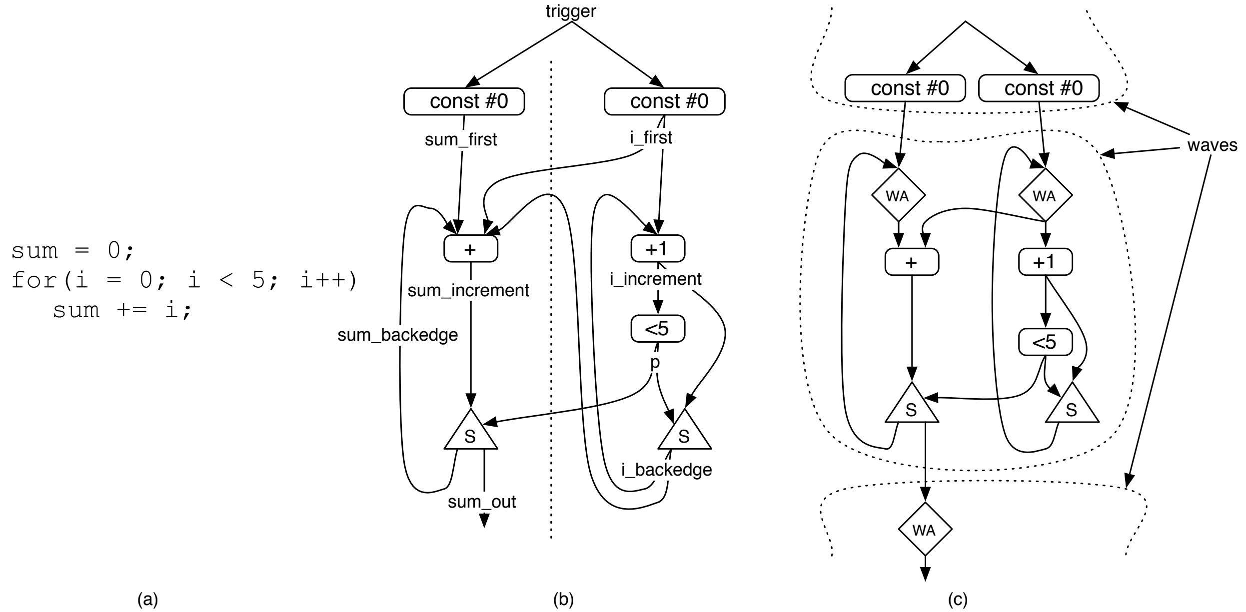

Figure 3: **Loops in WaveScalar:** A naive, slightly broken loop implementation (b), and the correct WaveScalar implementation (c).

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.

Being a dataflow architecture, WaveScalar must follow a slightly different convention. Since it has no registers, it does not need to preserve register values. It must, however, explicitly pass arguments and a return address to the function and trigger its execution. Passing arguments creates a data dependence between the caller and the callee. For indirect functions, these dependences are not statically known and therefore the static dataflow graph of the application does not contain them. Instead, WaveScalar provides a mechanism to send a data value to an instruction at a computed address. The instruction that allows this is called INDIRECT-SEND.

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.

Figures 4 contains the dataflow graph for a small function and a call site. Dashed lines in the graphs represent the dependences that exist only at run time. The LANDING-PAD instruction, as its name suggests, provides a target for a data value sent via INDIRECT-SEND. To call the function, the caller uses three INDIRECT-SEND instructions: 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 use the address of `foo` and their immediate values to compute the addresses of instructions they will send their values to.

When the values arrive at `foo`, the LANDING-PAD instructions pass them to WAVE-ADVANCE instructions that, in turn, forward them into the function body (the callee immediately begins a new wave). 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.

2.2.5 Memory ordering

Enforcing imperative languages memory semantics is one of the key challenges that have prevented dataflow processing from becoming a viable alternative to the von Neumann model. Since dataflow ISAs only enforce the static data dependences in a program's dataflow graph, they have no mechanism that ensures that memory operations occur in program order. Figure 5 shows a dataflow graph that demonstrates the dataflow memory ordering problem. In the graph the Load must execute

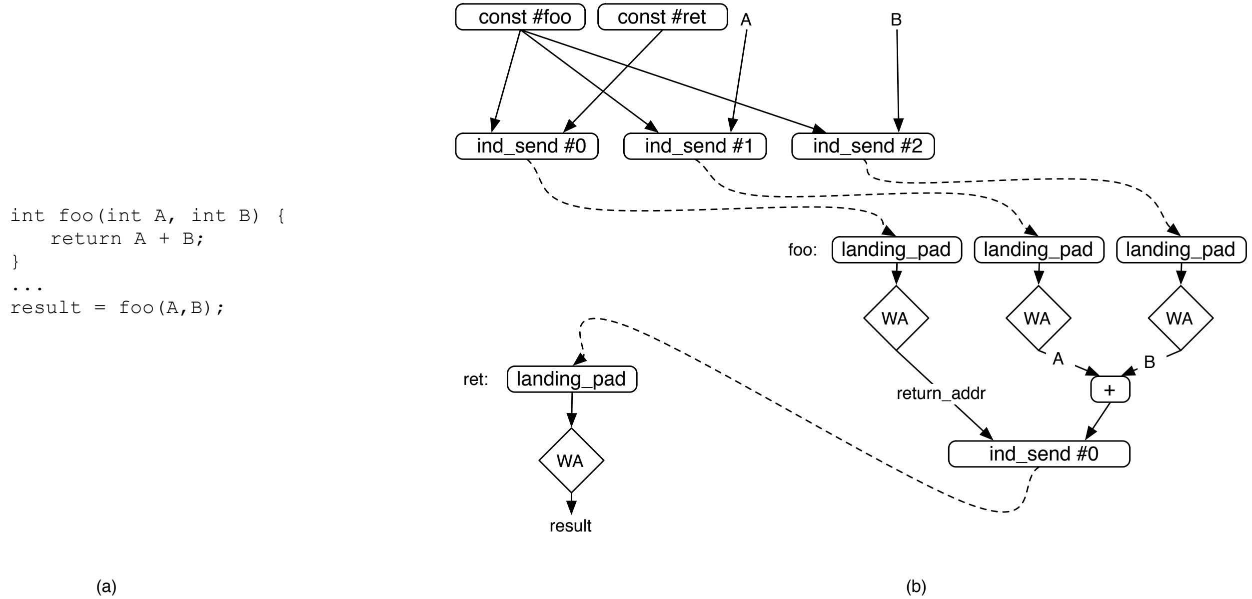

Figure 4: **A function call:** The dataflow graph (b) for a call to a simple function (a). The left-hand side of the dataflow graph uses INDIRECT-SEND instructions to call function `foo` on the right. The dashed lines show data dependences that WaveScalar must resolve at runtime. The immediate values on the trio of INDIRECT-SEND instructions are offsets from the first instruction in `foo`.

after the Store to ensure correct execution should the two memory addresses be identical. However, the dataflow graph does not express this implicit dependence between the two instructions (the dashed line). WaveScalar must provide an efficient mechanism to encode this implicit dependence in order to support imperative languages.

Wave-ordered memory solves the dataflow memory ordering problem, using the waves defined in Section 2.2.3. Within each wave, the compiler annotates memory access instructions to encode the ordering constraints between them. Since wave numbers increase as the program executes, they provide an ordering of the executing waves. Taken together, the coarse-grain ordering between waves (via their wave numbers), combined with the fine-grain ordering within each wave, provides a total order on all the memory operations in the program.

This section presents wave-ordered memory. Once we have more fully described waves and discussed the annotation scheme for operations within a wave, we describe how the annotations provide the necessary ordering. Then we briefly discuss an alternative solution to the dataflow memory ordering problem.

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

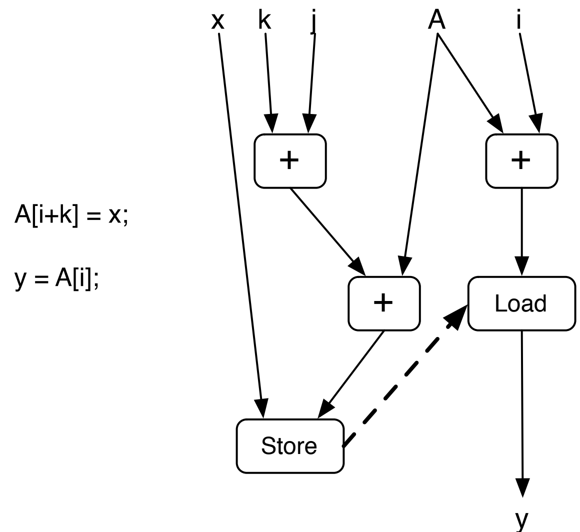

Figure 5: **Program order:** The dashed line represents an implicit, potential data dependence between the Store and Load instructions that conventional dataflow instruction sets have difficulty expressing. Without the dependence, the dataflow graph provides no ordering relationship between the memory operations.

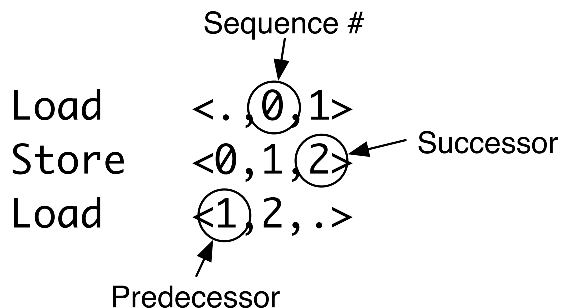

Figure 6: **Simple wave-ordered annotations:** The three memory operations must execute in the order shown. The predecessor, sequence, and successor numbers encode the ordering constraints. The '.' symbols indicate that operations 0 and 2 are the first and last operations in the wave.

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 if one memory operation has a larger sequence number than another, the one with the larger number comes later in program order. Figure 6 shows a very simple series of memory operations and their annotations. The sequence number is the second of the three numbers in angle brackets.

Second, wave-ordered memory must detect when all previous memory operations that will execute have done so. In the absence of branches, this detection is simple: since all the memory operations in a wave will eventually execute, the memory system simply waits for memory operations with all lower sequence numbers to complete. Control flow complicates this, because it allows some of the memory operations to execute (those on the taken paths) while others do not (those on the non-taken paths). To accommodate this, Wave-ordered memory must distinguish between operations that take a long time to fire and those that never will. To ensure that all the memory operations on the correct path are executed, each memory operation also carries the sequence number of its previous and subsequent operations in program order. Figure 6 includes these annotations as well. The predecessor number is the first number between the brackets, and the successor number is the last. For instance, the Store in the figure is preceded by a Load with sequence number 0 and followed by the Load with sequence number 2, so its annotations are $< 0, 1, 2 >$. The '.' symbols indicate that there is no predecessor of operation 0 and no successor of operation 2.

At branch (join) points the successor (predecessor) number is unknown at compile time, because control may take one of two paths. In these cases a 'wildcard' symbol, '?', takes the place of the successor (predecessor) number. The left-hand portion of Figure 7 shows a simple IF-THEN-ELSE control flow graph that demonstrates how the wildcard is applied; 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 path) executes each time the wave executes (i.e., during one dynamic instance of the wave). For instance, the right side of Figure 7 shows the sequence of operations along one path through the code on the left. From one operation to the next, either the predecessor and sequence numbers or the successor and sequence numbers match (the ovals in the figure).

In order 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 successor number '.', indicating there is no successor.

It is easy to enforce this condition on the beginning and the end of the chain of operations, but ensuring that all possible changes through the wave are complete is more difficult. Figure 8(a) shows an example. The branch and join mean that the instruction 0's successor and instructions 2's predecessor are both '?'. As a result, the memory system cannot construct the required chain between operations 0 and 2, if control takes the right-hand path. To create a chain, the compiler inserts a special MEMORY-NOP instruction between 0 and 2 on the right-hand path (Figure 8(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%.

Ordering Rules We can now demonstrate how WaveScalar use wave numbers and the annotations described above to construct a total ordering over all memory operations in a program. Figure 7 shows a simple example. Control takes the right-hand path resulting in three memory operations executing. At right, ovals show the links between the three operations that form them into a chain. The general rule is that a link ex-

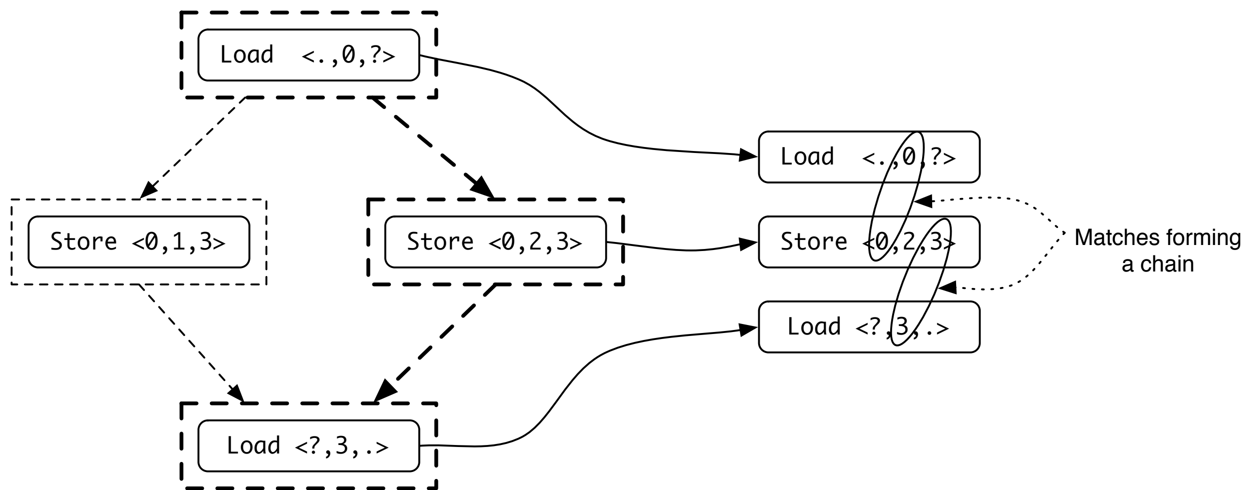

Figure 7: **Wave-ordering and control:** Dashed boxes and lines denote basic blocks and control paths. The right hand side of the figure shows the instructions that actually execute when control takes the right-hand path (bold lines and boxes) and the matches between their annotations that define program order.

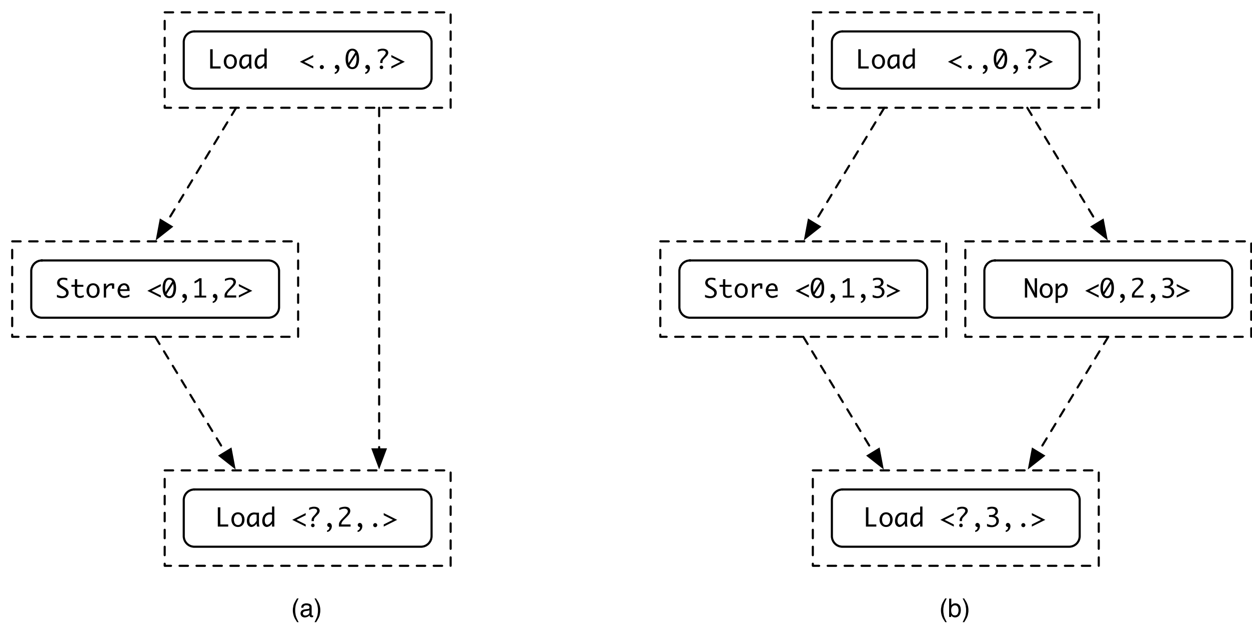

Figure 8: **Resolving ambiguity:** In (a), the chaining is impossible along the right-side path. In (b), the addition of a MEMORY-NOP allows chaining.

ists between two operations if the successor number of the first operation matches the sequence number of the second or the sequence number of the first matches the predecessor number of the second.

Since the annotations only provide ordering with a wave, WaveScalar uses wave numbers to order the waves themselves. The WaveScalar processor must ensure that all the operations from previous waves complete before the operations in a subsequent wave can be applied to memory. Combining the global inter-wave ordering with the local intra-wave ordering provides a total ordering on all operations in the program.

Expressing parallelism The basic version of wave-ordered memory described above can be easily extended to express parallelism between memory operations, allowing consecutive Loads to execute in parallel or out-of-order.

The annotations and rules define a linear ordering of memory operations, ignoring potential parallelism between Loads. Wave-ordered memory can express this parallelism by providing a fourth annotation called a *ripple number*. The ripple number of a Store is equal to its sequence number. A Load's ripple number points to 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 9 shows a sequence of Load and Store operations with all four annotations.

To accommodate ripples in the ordering rules we allow a Load to execute if it is next in the chain operations (as before), *or* if the ripple number of the Load is less than or equal to the sequence number of a previously executed operation (a Load or a Store). MEMORY-NOPS are treated like Loads.

Figure 9 shows the two different types of links that can allow an operation to fire. The solid oval between the bottom four operations are similar to those in Figure 7. The top two dashed ovals depict ripple-based links that allow the two Loads to execute in parallel.

Figure 10 contains a more sophisticated example. If control takes the right-side branch, Loads 1 and 4-6 can execute in parallel once Store 0 has executed, because they all have ripple numbers of 0. Load 7 must wait for one of Loads 4-6 execute, because the ripple number of operation 7 is 2 and Loads 4-6 all have sequence numbers greater than 2. If control takes the left branch, Loads 3 and 7 can execute as soon as Store 2 has exe-

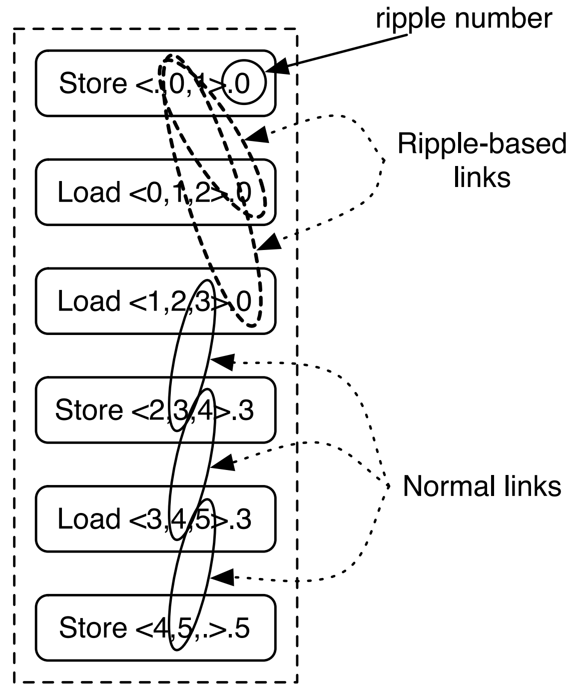

Figure 9: **Simple ripples:** A single wave containing a single basic block. The ripple annotations allow loads 1 and 2 to execute in either order or in parallel, while the stores must wait for all previous loads and stores to complete. Ovals depict the links formed between operations.

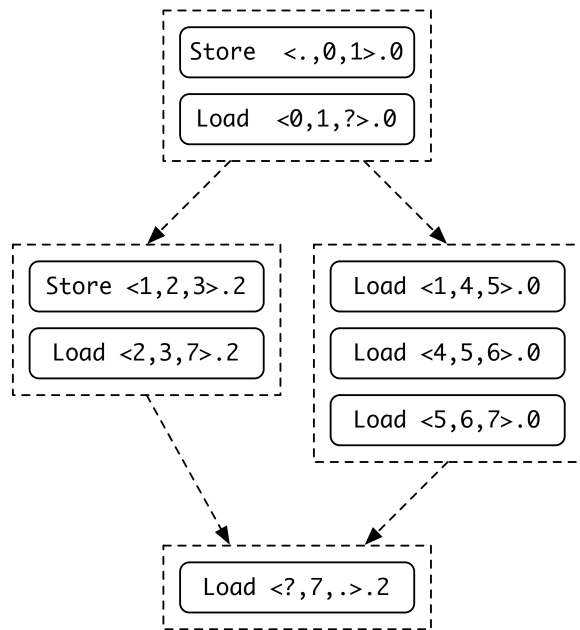

Figure 10: **Ripples and control:** Branches make ripple behavior more complicated. If control takes the right-hand path, most of loads (1, and 4-6) can execute in any order, but load 7 must wait for an operation with a sequence number greater than 2.

cuted.

2.2.6 Other approaches

Wave-ordered memory is not the only way to provide the required memory ordering. Researchers have proposed an alternative scheme that makes implicit memory dependences explicit by adding a dataflow edge between each memory operation and the next [14, 15]. While this "token-passing" scheme is simple, it does not perform as well as wave-ordered memory; our experiments have found that wave-ordered memory expresses twice as much memory parallelism as token passing [16].

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 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 (Section 6).

2.3 Discussion

The WaveScalar instruction set this section describes is sufficient to execute single-threaded applications written in conventional imperative programming languages. The instruction set is slightly more complex than a conventional RISC ISA, but we have not found the complexity difficult for the programmer or the compiler to handle.

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. Only instructions that must pass each other data communicate. 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.

3 A WaveCache architecture for single-threaded programs

WaveScalar's overall goal is to enable an architecture that avoids the scaling problems described in the introduction. With the decentralized WaveScalar ISA in hand, our task is to develop a decentralized, scalable architecture to match. In addition to scaling challenges, the WaveCache also must address additional challenges specific to WaveScalar. The WaveCache must efficiently implement the dataflow firing rule and provide storage for multiple (perhaps many) instances of data values with different tags. It must also provide an efficient hardware implementation of wave-ordered memory.

This section describes a tile-based WaveScalar architecture, called the *WaveCache*, that addresses these challenges. The WaveCache comprises everything, except main memory, required to run a WaveScalar program. It contains a scalable grid of simple, identical dataflow processing elements that are organized hierarchically to reduce operand communication costs. 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 resulting architecture directly addresses two of the challenges we outlined in the introduction. First, the WaveCache contains no long wires. In particular, as the size of the WaveCache increases, the length 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 multithreaded workloads (See Section 5). The larger designs contain more tiles, but the tile structure, and therefore, the overall design complexity does not change. The final challenge mentioned in the introduction, 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 3.1. Next, Sections 3.2 to 3.6 provide a more detailed description of its major components and how they interact. Section 3.7 describes a synthesizable RTL model that we use, in combination with simulation studies, to provide the specific architectural parameters for the WaveCache we describe. Section 4 evaluates the design in terms of performance and the amount of area it requires.

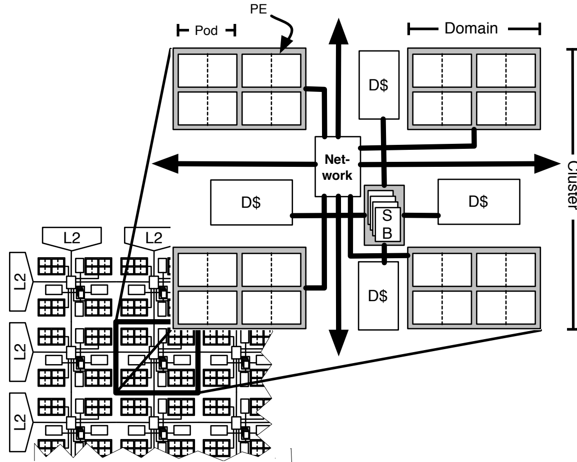

Figure 11: **The WaveCache:** The hierarchical organization of the microarchitecture of the WaveCache.

3.1 WaveCache architecture overview

Several recently proposed architectures, including the WaveCache, take a tile-based approach to addressing the scaling problems outlined in the introduction [17, 18, 19, 20, 21, 15]. Instead of designing a monolithic core that comprises the entire die, tiled processors cover the die with hundreds or thousands of 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, tiles more quickly amortize design and verification costs. Tiled architectures also 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 11). A cluster contains four identical *domains*, each with eight identical processing elements (PEs). In addition, each cluster has a four-banked L1 data cache, wave-ordered memory interface hardware, and a network switch for communicating with adjacent clusters.

From the programmer's perspective, every static instruction in a WaveScalar binary has a dedicated processing element. Clearly, building an array of clusters large enough to give each instruction in an entire application its own PE is impractical and wasteful, so, in practice, we dynamically bind multiple instructions to a fixed number of PEs, each of which can hold up to 64 instructions. Then, 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

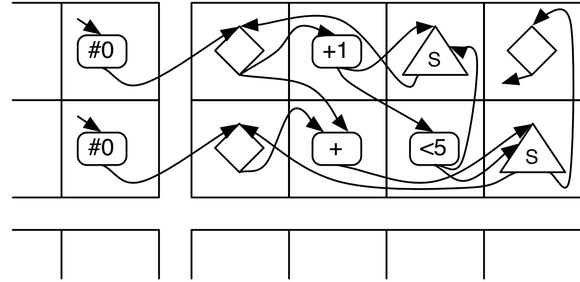

Figure 12: **Mapping instruction into the WaveCache:** The loop in Figure 3(c) mapped onto two WaveCache domains. Each large square is a processing element.

moniker.

Instructions are mapped to and placed in PEs dynamically as a program executes. The mapping algorithm has two often conflicting goals: to place dependent instructions near each other (e.g., in the same PE) to minimize producer-consumer operand latency, and to spread independent instructions out across several PEs to exploit parallelism. Figure 12 illustrates how the WaveScalar program in Figure 3(c) can be mapped into two domains in the WaveCache. To minimize operand latency, the entire loop body has been placed 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, associative tag matching store found on some previous dataflow machines [9]. Second, although PEs dynamically schedule execution, the scheduling hardware is dramatically simpler than a conventional dynamically scheduled processor. Section 3.2 describes the PE design in more detail.

To reduce communication costs within the grid, PEs are organized hierarchically along with their communication infrastructure (Figure 11). 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 buses. The four domains in a cluster communicate 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 memory interface in their local cluster. If possible, the local L1 cache provides the data. 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 with a possibly high WaveCache miss rate 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 protocol with multiple readers and single writer maintains cache coherence.

3.2 The PE

At a high level, the structure of a PE pipeline resembles a conventional five-stage, dynamically scheduled execution pipeline. The biggest difference between the two 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 main goal in designing the PE was to meet our cycle-time goal while still allowing dependent instructions to execute on consecutive cycles. Pipelining was relatively simple. Back-to-back execution, however, was the source of significant complexity.

The PE's pipeline stagers 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 [9]. 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 [9, 8].

The matching table is associated with a second, smaller *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, the 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 13 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 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 in-

structions 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 13) 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, B is inserted 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 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.

3.3 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 14). There are 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 subsections.

3.3.1 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, i.e., they have separate matching tables, scheduling and output queues, etc.

The decision to integrate pairs of PEs together is a response to two competing concerns: we wanted 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) had to be pipelined. This increased average communication latency and reduced performance significantly. Allowing pairs of PEs to communicate quickly brought 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 were small.

3.3.2 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 a normal PE's.

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.

3.3.3 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.

3.3.4 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, each of which 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.

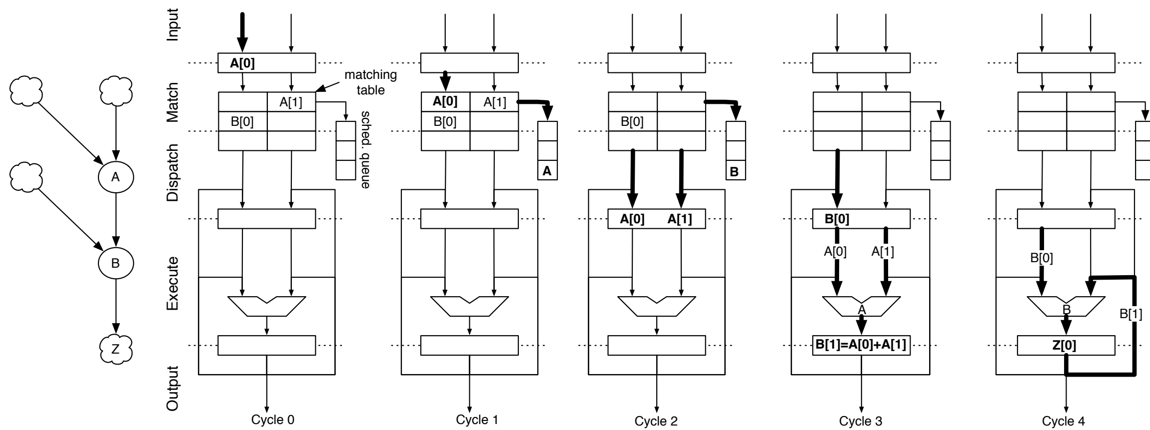

Figure 13: **The flow of operands through the PE pipeline and forwarding networks:** The figure is described in detail in the text.

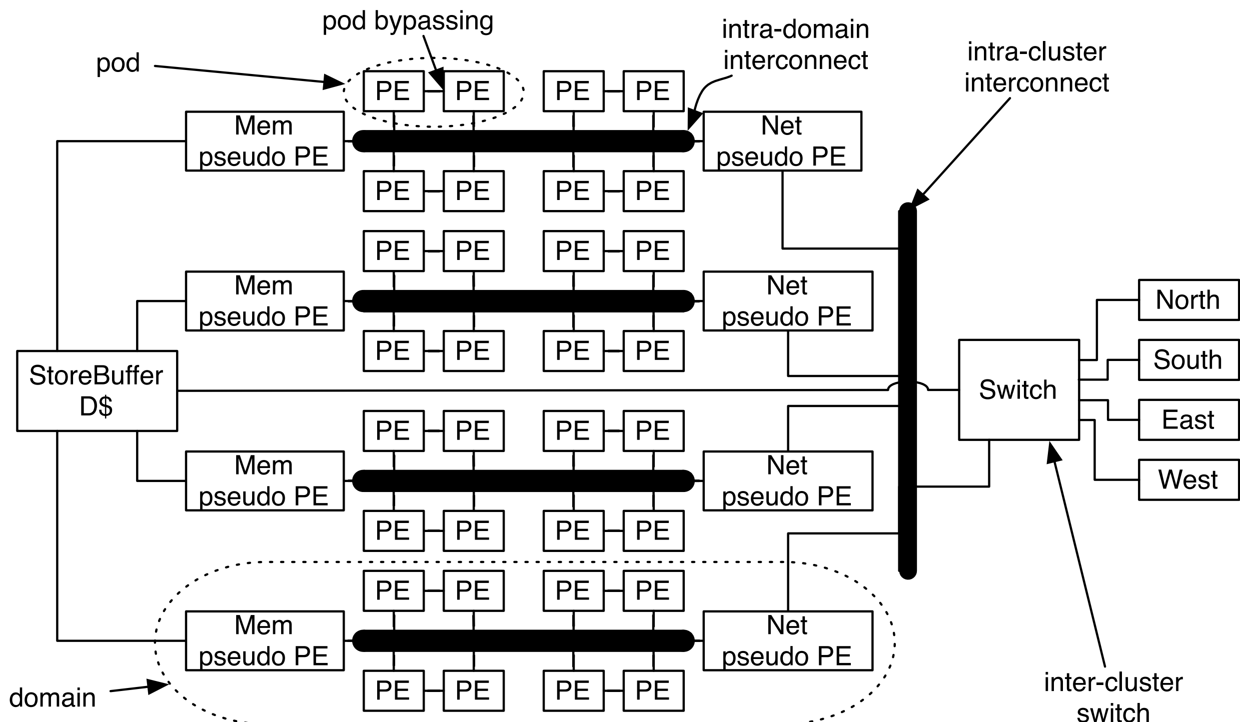

Figure 14: **The cluster interconnects:** A high-level picture of a cluster illustrating the interconnect organization.

Each input/output port supports the transmission of up to two operands. Its routing follows a simple protocol: the current buffer storage state at each switch is sent to the adjacent switches, which receive this information a clock cycle later. Adjacent switches only send information if the receiver is guaranteed to have space.

The inter-cluster switch provides two virtual channels that the interconnect uses to prevent deadlock [22]. 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.

3.4 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 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 buffer via the inter-cluster interconnect. 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 $\mathrm{pred}(R)$, $\mathrm{seq}(R)$, and $\mathrm{succ}(R)$ as the wave-ordering annotations for a request R. We also define $\mathrm{next}(R)$ to be the sequence number of the operation that actually follows R in the current instance of the wave. $\mathrm{next}(R)$ is determined either directly from $\mathrm{succ}(R)$ or is calculated by the wave-ordering hardware, if $\mathrm{succ}(R)$ is '?'.

The store buffer 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 fire 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 $\mathrm{seq}(R)$. If $\mathrm{succ}(R)$ is defined (i.e., not '?'), the entry in the next table at location $\mathrm{seq}(R)$ is updated to $\mathrm{succ}(R)$. If $\mathrm{pred}(R)$ is defined, the entry in the next table at location $\mathrm{pred}(R)$ is set to $\mathrm{seq}(R)$.

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. 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 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. 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.

3.5 Caches

The rest of the WaveCache's memory hierarchy comprises a 32KB, four-way set associative L1 data cache at each cluster, and a 16MB L2 cache distributed along the edge of the chip (16 banks in a 4x4 WaveCache). A

directory-based multiple reader, 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.

3.6 Placement

Placing instructions carefully into the WaveCache is critical to good performance, because of the competing concerns we mentioned earlier. Instructions' proximity determines the communication latency between them, arguing for tightly packing instructions together. On the other hand, instructions that can execute simultaneously should not end up at the same processing element, 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 [23]. Here, we describe the approach we used for the studies in this paper.

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 and consecutive 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 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.

This 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. We use a well-known dataflow technique, k-loop bounding [24], to restrict the number iterations, k, that can be executing at one time. We tune k for each application.

3.7 The RTL model

To explore the area, speed, and complexity implications of the WaveCache architecture, we have developed a synthesizable RTL model of the components described above. We use the RTL model, combined with detailed architectural simulation, to tune the WaveCache's parameters and make trade-offs between performance, cycle time, and silicon area. All the specific parameters of the architecture (e.g., cache sizes, bus widths, etc.) we present reflect the results of this tuning process. The design we present is a WaveCache appropriate for general purpose processing in 90nm technology. Other designs targeted at specific workloads or future process technologies would differ in choice of particular parameters, but the overall structure of the design would remain the same. A thorough discussion of the RTL design and the tuning process is beyond the scope of this paper (but can be found in [25]). Here, we summarize the methodology and the timing results.

We derive our results with the design rules and the recommended tool infrastructure of the Taiwan Semiconductor Manufacturing Company's TSMC Reference Flow 4.0 [26], which is tuned for 130nm and smaller designs (we use 90nm). By using these up-to-date specifications, we ensure, as best as possible, that our results scale to future technology nodes. To ensure that our measurements are reasonable, we follow TSMC's advice and feed the generated netlist into Cadence Encounter for floorplanning and placement, and then use Cadence NanoRoute for routing [27]. After routing and RC extraction, we measure the timing and area values.

According to the synthesis tools, our RTL model meets our timing goal of a 20 FO4 cycle time ($\sim$1Ghz in 90nm). The cycle time remains the same regardless of the size of the array of clusters. The model also provides detailed area measurements for the WaveCache's components. Table 1 shows a break down of area within a single cluster. The ratios for an array of clusters are the same.

In the next section we evaluate the WaveCache's per-

Component	Fraction of cluster area
PE stages	
INPUT	7%
MATCH	22%
DISPATCH	38%
EXECUTE	4%
OUTPUT	7%
PE total	78%
inter-cluster interconnect switch	2%
storebuffer	17%
L1 cache	3%

Table 1: **A cluster's area budget:** A breakdown of the area required for a cluster. Most of the area is devoted to processing resources.

formance on single-threaded applications and compare its performance and area requirements with a conventional superscalar processor.

4 Single-threaded WaveCache performance

This section measures the WaveCache's performance on a variety of single-threaded workloads. We measure the performance of a single-cluster WaveCache design using cycle-accurate simulation of the architecture in Section 3. This WaveCache achieves performance that is similar to that of a conventional out-of-order superscalar processor, but does so in only **30**% as much area.

Before we present the performance results in detail, we review the WaveCache's parameters and describe our workloads and toolchain.

4.1 Methodology

Table 2 summarizes the parameters for the WaveCache we use in this section.

To evaluate WaveCache performance, we use an execution-driven, cycle accurate simulator that closely matches our RTL model. The performance we report here is lower than that in the original WaveScalar paper [2]. The discrepancy is not surprising, since that work used an idealized memory system (perfect L1 data caches), larger, 16-PE domains, and a non-pipelined design.

In the experiments in this section, we use nine benchmarks from three groups. From SpecINT2000: *gzip*, *mcf*, *twolf*; from SpecFP2000: *ammp*, *art*, *equake*; and from Mediabench: *djpeg*, *mpeg2encode*, *rawdaudio*. We compiled each application with the DEC cc compiler using `-O4 -fast -inline speed` optimizations. A binary translator-based toolchain 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 binary translator. The binary translator cannot process some programming constructs (e.g., compiler intrinsics that don't obey the alpha calling convention and jump tables), but this is strictly a limitation of our translator, not a limitation of WaveScalar's ISA or execution model. We are currently working on a full-fledged compiler that will allow us to run a wider range of applications.

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 [28]. AIPC measures the number of non-overhead instructions (e.g., STEER, ϕ, etc.) executed per cycle. The AIPC measurements for the superscalar architectures we compare to are in good agreement with other measurements [29].

After the startup portion of each application is finished, we run each application for 100 million Alpha instructions, or to completion.

4.2 Single threaded performance

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 measurements, we use `sim-alpha` configured to model the Alpha EV7 [30, 31], but with the same L1, L2, and main memory latencies we model for the WaveCache. The two WaveCache configurations are *WC1x1*, a 1x1 array of clusters, and *WC2x2*, a 2x2 array. The only other difference between the two is the size of the L2 cache (1MB for WC1x1 vs 4MB for WC2x2).

Figure 15 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 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

WaveCache Capacity	2K(WC1x1) or 8K(WC2x2) static instructions (64 per PE)		
PEs per Domain	8 (4 pods)	Domains / Cluster	4
PE Input Queue	16 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 Caches	32KB, 4-way set associative, 128B line, 4 accesses per cycle	L2 Cache	1MB (WC1x1) or 4MB (WC2x2) shared, 128B line, 16-way set associative, 10 cycle access
Main RAM	200 cycle latency	Network Switch	2-port, bidirectional

Table 2: Microarchitectural parameters of the baseline WaveCache

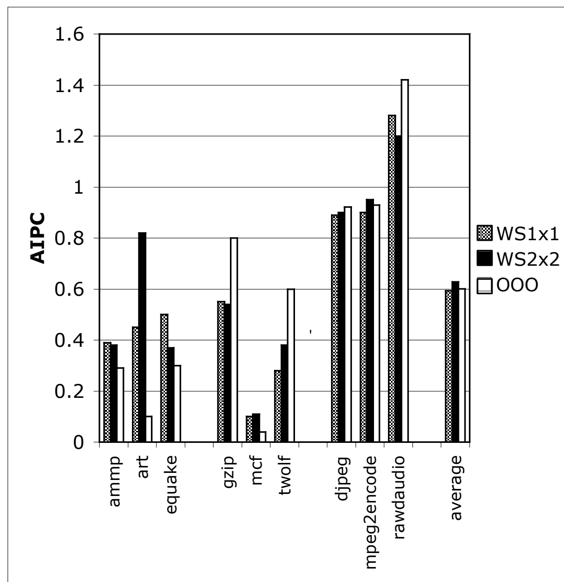

Figure 15: **Single-threaded WaveCache vs. superscalar:** On average, both WaveCaches perform comparably to the superscalar.

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 tends to do 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), and two factors contribute to WaveScalar's superior performance. First, WaveScalar's dataflow execution model allows several 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 reduc-

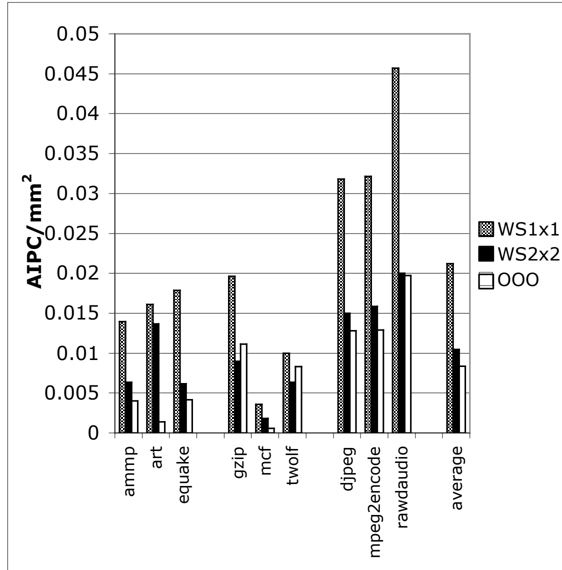

Figure 16: **Performance per unit area:** The 1x1 WaveCache is the clear winner in terms of performance per area.

ing 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 [31, 32]. 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 PLL, 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 28mm^2 (12mm^2 of L2 cache) and WC2x2 occupies 100mm^2 (44mm^2 of L2 cache) in 90nm.

Figure 16 shows the area-efficiency of the WaveCaches measured in AIPC/mm^2 compared to OOO. The WaveCache's more compact design allows WS1x1 to extract 2.5$\times$ as much AIPC per area as OOO, on average. The results for WS2x2 show that, for these applications, quadrupling the size of the WaveCache does not have an commensurate effect on performance.

Because OOO is configured to match the EV7, it has twice as much on-chip cache as WS1x1. To measure the effect of the extra memory, we halved the amount of cache in the OOO configuration (data not shown). This change reduced OOO's area by 41% and its performance by 17%. WS1x1 provides 80% more performance per area than this configuration.

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. In the next two sections we extend WaveScalar's abilities to handle conventional pthread-style threads and to exploit its dataflow underpinnings to execute fine-grain threads. In these areas, the WaveCache's performance is even more impressive.

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 that WaveScalar must address 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 is tightly integrated into the dataflow framework.

The required changes to the WaveCache to support the ISA extensions are surprisingly small, and do not

impact the overall structure of the WaveCache, because executing threads dynamically share most WaveCache processing resources.

To evaluate the WaveCache's multithreaded performance, we simulate an 64-cluster design, representing an aggressive "big iron" processor built in next-generation process technology and suitable for large-scale multithreaded programs. For most Splash-2 benchmarks, the WaveCache achieves nearly linear speedup with up to 64 concurrent threads. 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 WaveCache outperforms the von Neumann designs by a factor of between 2 and 16.

The next two sections describe the multihthreading ISA extensions. Section 5.3 presents the Splash-2 results and contains the comparison to multithreaded von Neumann machines.

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 architecture 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 paper, 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 $<t_1, w_1>.d$. 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-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 17 illustrates how, using the new instructions, thread t creates a new thread s, thread s executes and then terminates.

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

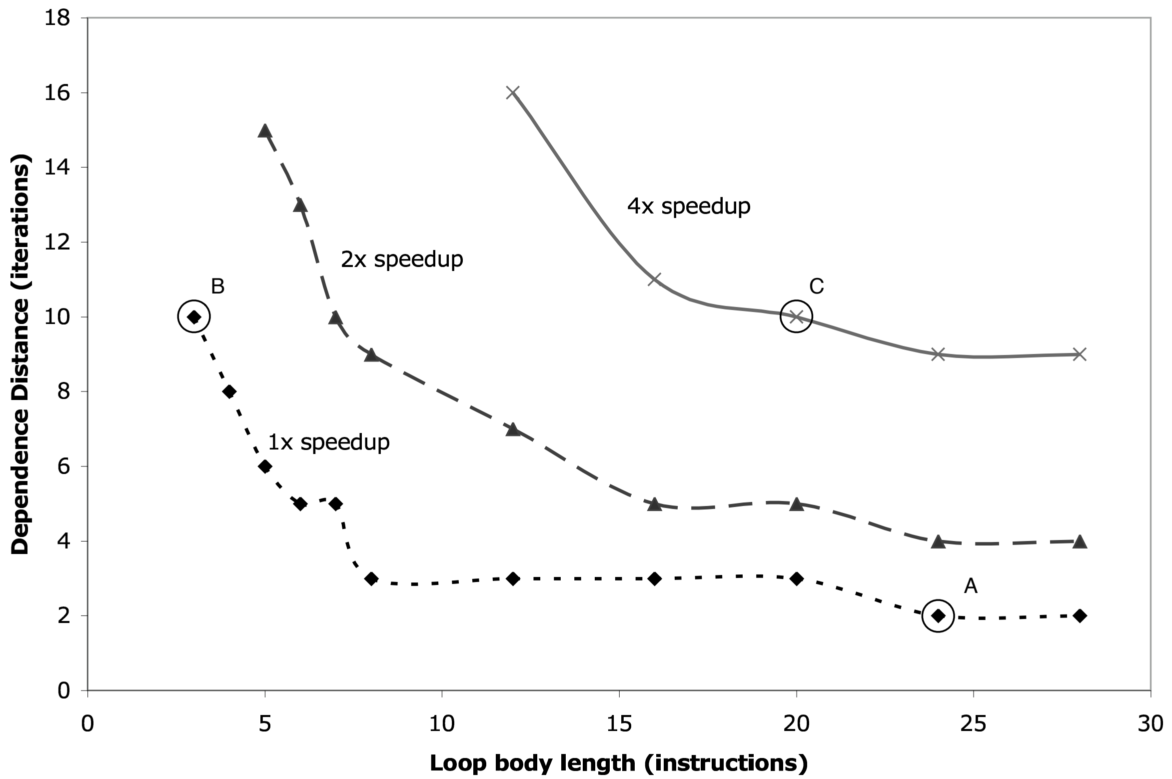

Figure 18: **Thread creation overhead:** Contour lines for speedups of $1\times$ (no speedup), $2\times$ and $4\times$. The area above the each line is a region of program speedup at or above the stated value. Spawning wave-ordered threads in the WaveCache is lightweight enough to profitably parallelize loops with as few as ten instructions in the loop body if four independent iterations may execute.

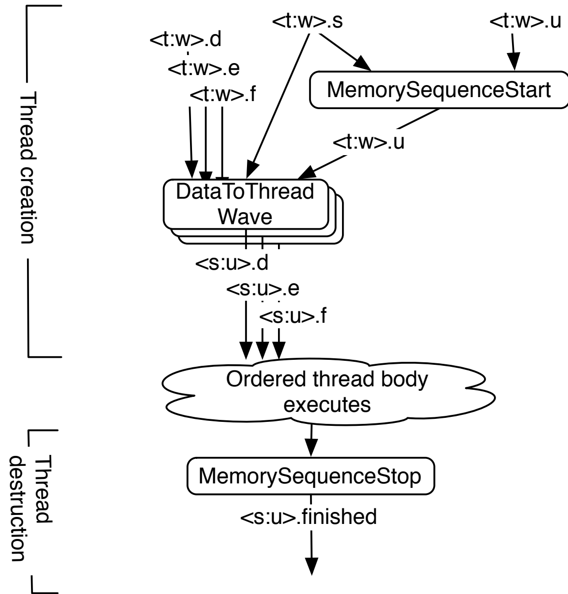

Figure 17: **Thread creation and destruction:** Thread t spawns a new thread s by sending s's THREAD-ID (s) and WAVE-NUMBER (u) to MEMORY-SEQUENCE-START, which allocates a store buffer to handle the first wave in the new thread. The result of the MEMORY-SEQUENCE-START instruction helps trigger the three 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.

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 was 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 18 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 subsection, provides only part of the functionality required to make multithreading 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 multithreaded 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 [33]. 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 multithreaded 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 [34, 35]. 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 [36, 37], but is tailored to WaveScalar's dataflow framework.

As Figure 19 illustrates, THREAD-COORDINATE requires slightly different matching rules. [2] All WaveScalar instructions *except* THREAD-COORDINATE fire when the tags of two input values match, and they produce outputs with the same tag (Figure 19, 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 19, 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 19, 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 microarchitectural counters at each PE to relabel the WAVE-NUMBERs of the inputs to THREAD-COORDINATE instructions so they 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, for brevity we only illustrate a simple mutex (Figure 20). 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 20 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

[2] 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 [38]. Monsoon's M-structure store units had a special matching rule to enforce load-store order [39].

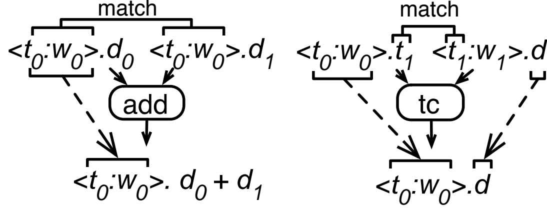

Figure 19: **Tag matching:** Most instructions, like the ADD shown here at left, fire when the thread and wave numbers on both input tokens match. Inputs to THREAD-COORDINATE (right) match if the THREAD-ID of the token on the second input matches the data value of the token on the first input.

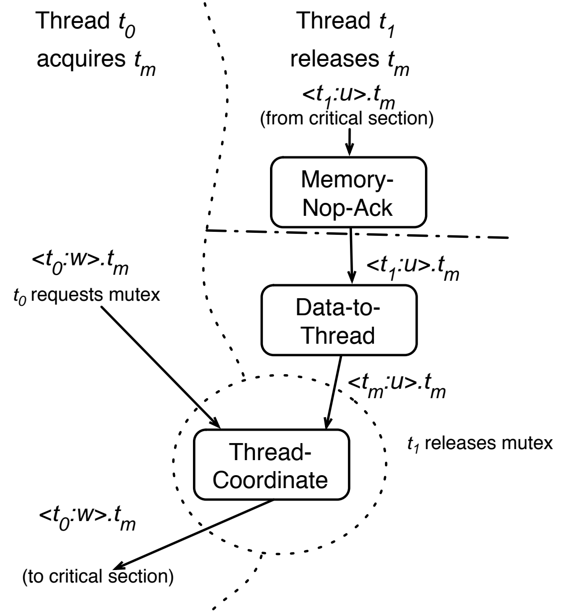

Figure 20: **A mutex:** THREAD-COORDINATE is used to construct a mutex, as described in the text.

Benchmark	Parameters
fft	-m12
lu	-n128
radix	-n16384 -r32
ocean-noncont	-n18
water-spatial	64 molecules

Table 3: Splash-2 benchmarks and their parameters used in this study.

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 Splash-2

In this section, we evaluate WaveScalar's multithreading facilities by executing coarse-grain, multithreaded applications from the Splash-2 benchmark suite (Table 3). We use the toolchain and simulator described in Section 4.1. We simulate an 8x8 array of clusters to model an aggressive, future-generation design. Using the results from the RTL model described in Section 3.7 scaled to 45nm, we estimate that the processor occupies $\sim 290\text{mm}^2$, with an on-chip 16MB L2.

After skipping past initialization, we measure execution of the parallel phases of the benchmarks. 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 help explain these bottom-line results, where appropriate.

Evaluation of a multithreaded WaveCache. Figure 21 contains speedups of multithreaded WaveCaches for all six benchmarks, as compared to their single-threaded running time. 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

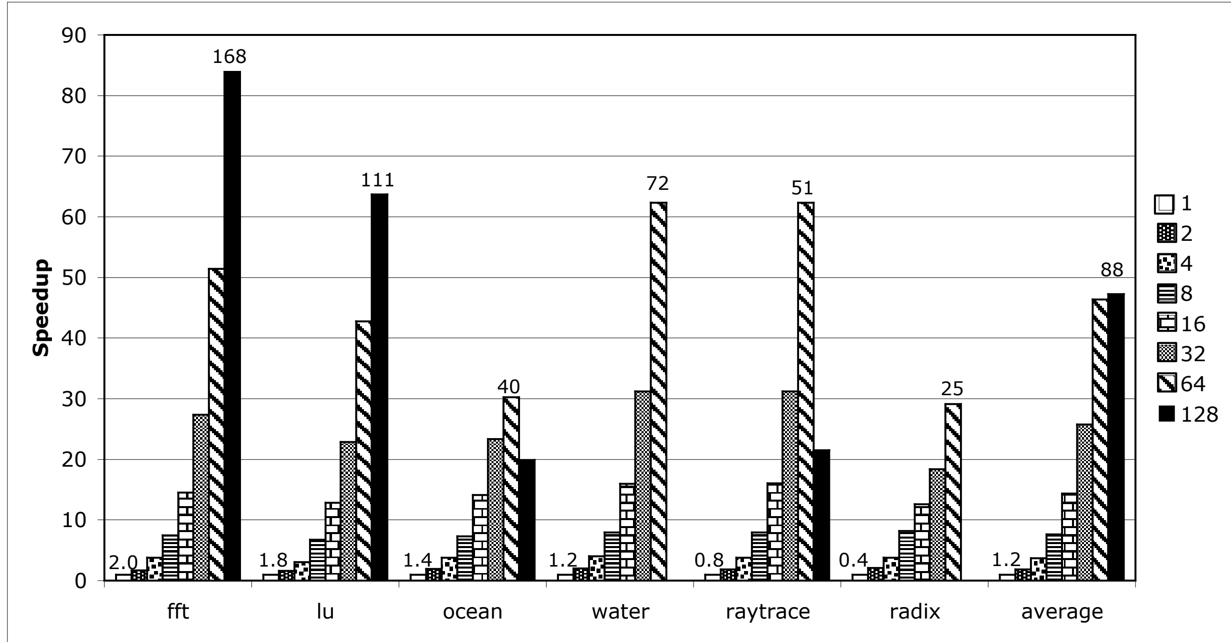

Figure 21: **Splash-2 on the WaveCache.:** We evaluate each of our Splash-2 benchmarks on the baseline WaveCache with between 1 and 128 threads. The bars represent speedup in total execution time. The numbers above the single-threaded bars are IPC for that configuration. Two benchmarks, *water* and *radix*, cannot utilize 128 threads with the input data set we use, so that value is absent.

compulsory misses. In addition, the operand matching cache miss rate increases by 23%. Finally, the data cache miss rate, which 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 threaded architectures We compare the performance of the WaveCache and a few other architectures on three Splash-2 kernels: *lu*, *fft* and *radix*. 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 [41]. Comparing data from such diverse sources is difficult, and drawing precise conclusions about the results is hard; however, we believe that the measurements are still 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 22 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

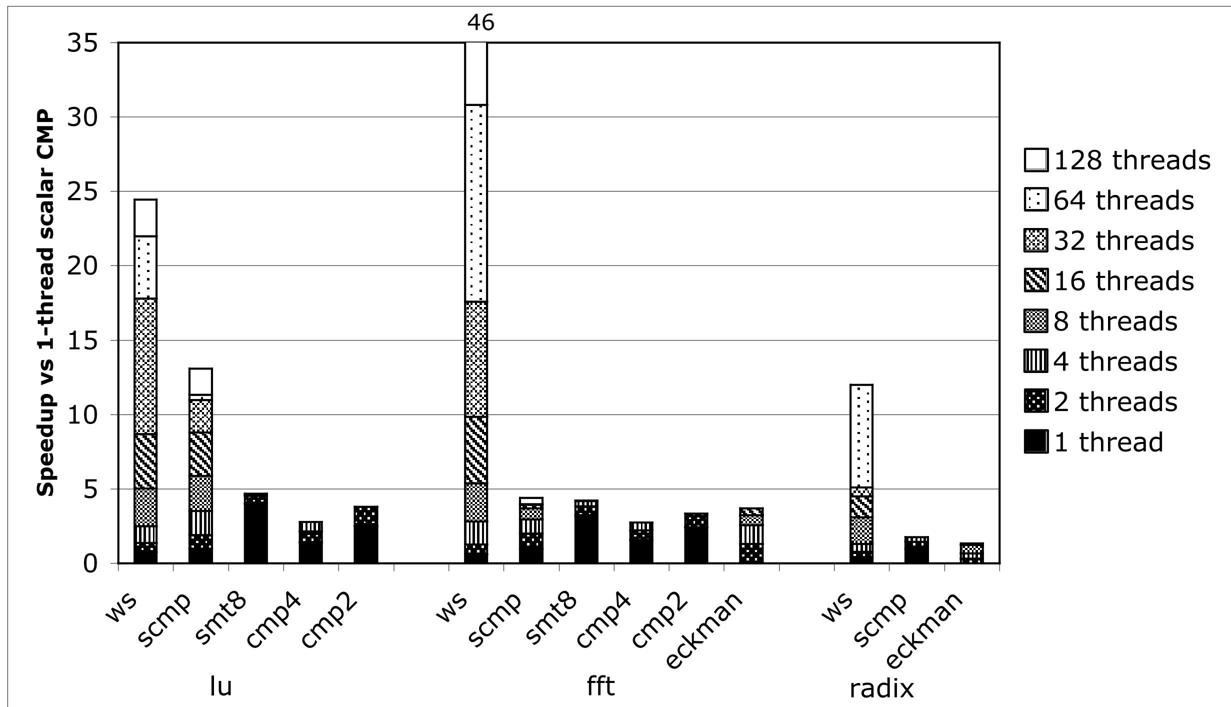

Figure 22: **Performance comparison of various architectures.:** Each bar represents performance of a given architecture for a varied number of threads. We normalize running times to that of a single-issue scalar processor with a high memory access latency, and compare speedups of various multithreaded architectures. *ws* is a 4x4 WaveCache. *scmp* is a CMP of the aforementioned scalar processor on a shared bus with MESI coherence. *smt8*, *cmp4* and *cmp2* are an 8-threaded SMT, a 4-core out-of-order CMP and a 2-core OOO CMP with similar resources, from [40]. *ekman* [41] is a study of CMPs in which the number of cores is varied, but the number of execution resources (functional units, issue width, etc.) is fixed.

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 coherence is maintained by a 4-state snoopy protocol. 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* [42] and a single *Piranha* chip [43], 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 [41] (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 [41], 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 graph shows that the WaveCache can handily outperform the other architectures at high thread counts. It executes $1.8\times$ to $10.9\times$ faster than *scmp*, $5.2\times$ to $10.8\times$ faster than *smt8*, and $6.4\times$ to $16.6\times$ 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[3]. 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 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.

5.4 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 of von Neumann-style alternatives, and, as we mentioned earlier, scaling the WaveCache to future process technologies is straightforward. 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.

In the next section we investigate the potential of WaveScalar's core dataflow execution model to support a second, finer-grain threading model. These fine-grain threads utilize a simpler, unordered memory interface, and can provide huge performance gains for some applications.

6 WaveScalar's dataflow side

The WaveScalar instruction set 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 if WaveScalar is to be a viable alternative to von Neumann architectures, but it is not the limit of what WaveScalar can do.

This section exploits WaveScalar's dataflow underpinning to achieve two things that conventional von

[3]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.

Neumann machines cannot. First, it provides a second, *unordered* memory interface that is similar in spirit to the token-passing interface in Section 2.2.6. 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. As we describe below, the programmer can restrict this ordering by adding edges to the programs dataflow graph.

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 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, but any number of threads can use the unordered interface at one time. In addition, spawning these threads is very cheap. As a result, 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 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 $9\times$ speedup.

6.1 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.

6.1.1 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 Section 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*)

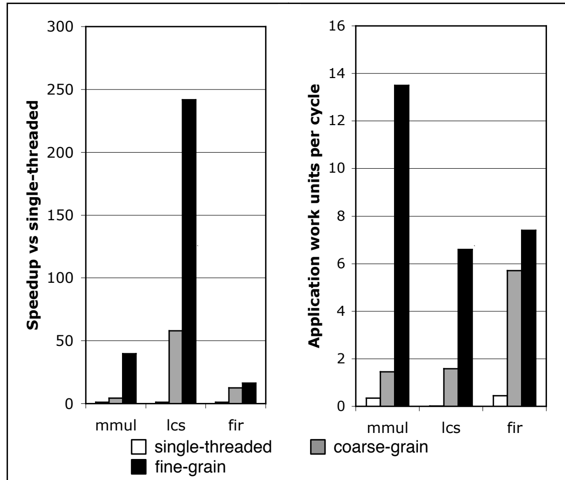

Figure 23: **Fine-grain performance:** These graphs compare the performance of our three implementation styles. The graph on the left shows execution-time speedup relative to the serial coarse-grain implementation. The graph on the right compares the work per cycle achieved by each implementation measured in multiply-accumulates for MMUL and FIR and in character comparisons for LCS.

threads. Each version is run to completion.

Figure 23 depicts the performance of each algorithm executing on the 8x8 WaveCache described in Section 5.3. On the left, it shows speedup over the serial implementation, and, on the right, average units of work completed per cycle. For MMUL 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, within each thread 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 only once pass and then pass them from thread to thread directly. For these two applications the limiting factor is inter-cluster network bandwidth. Both algorithms involves 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 better placement algorithm could alleviate much of this problem and improve performance further by placing the instructions for communicating threads near one another.

6.2 Mixing threading models

In Section 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.

6.2.1 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. In this section, we take a detailed look at how this is achieved.

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 interfaces to exist simultaneously.

Example 1: Figure 24 shows a code sequence that transitions from wave-ordered memory to unordered

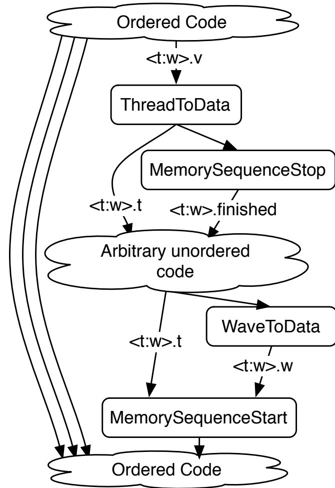

Figure 24: **Transitioning between memory interfaces:** The transition from ordered to unordered memory and back again.

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 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 25 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 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.

6.2.2 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 [44] 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 *equake*'s overall performance by a factor of 1.6.

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

```
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;
}
```

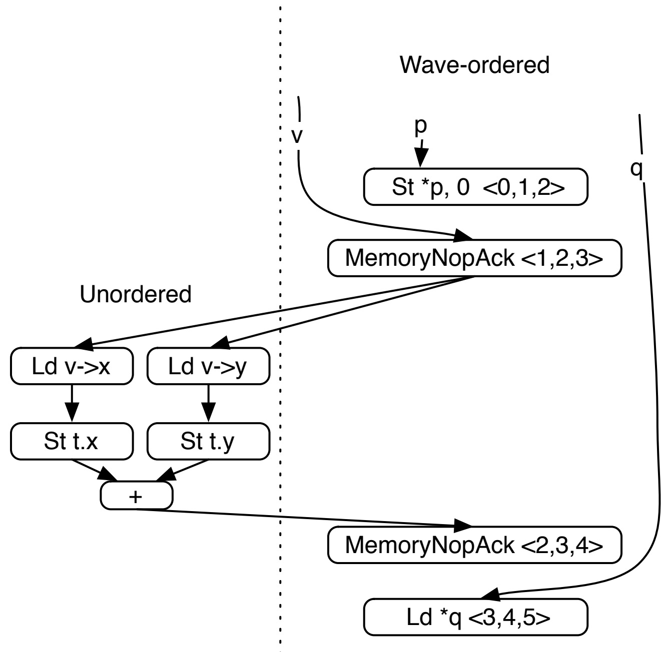

Figure 25: **Using ordered and unordered memory together:** A simple example where MEMORY-NOP-ACK is used to combine ordered and unordered memory operations to express memory parallelism.

a particular value at a time. This idiom is inspired by M-structures [37], a dataflow-style memory element. Rewriting *smvp* with unordered memory and THREAD-COORDINATE improves overall performance by a factor of 7.9.

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.

7 Conclusion

The WaveScalar instruction set and WaveCache architecture demonstrate that dataflow processing is a worthy alternative to the von Neumann model and conventional scalar designs for both single- and multi-threaded workloads.

Like all dataflow ISAs, WaveScalar allows programmers and compilers to explicitly express parallelism among instructions. Unlike previous dataflow models, WaveScalar also includes a memory-ordering scheme, wave-ordered memory, that allows it to efficiently execute programs written in conventional, imperative programming languages.

WaveScalar's multithreading facilities support a range of threading styles. For conventional pthread-style threads, WaveScalar provides thread creation and termination instructions, multiple, independent wave-ordered memory orderings, and a lightweight, memoryless synchronization primitive, and a memory fence that provides a relaxed consistency model. For finer threads, WaveScalar can disable memory ordering for specific memory accesses, allowing the programmer or compiler to express large amounts of memory-parallelism, and enabling a very fine-grain style of multithreading. Finally WaveScalar allows both types of threads to coexist in a single application and interact smoothly.

The WaveCache architecture exploits WaveScalar's decentralized execution model to eliminate broadcast communication and centralized control. Its tile-based designs makes it scalable and significantly reduces the architecture's complexity. Our RTL model shows that a WaveCache capable of running real-world, multi-threaded applications would occupy only 253mm^2 in currently available process technology, while a single threaded version requires only 28mm^2.

Our experimental results show that the WaveCache performs comparably to a modern out-of-order design on average for single threaded codes. For multithreaded, Splash2 benchmarks, the WaveCache achieves 30-83$\times$ speedup over a single threaded versions, and outperforms a range of von Neumann-style multithreaded processors by a wide margin. By exploiting our new unordered memory interface, we demonstrated how hundreds of fine-grain threads on the WaveCache can com-

plete up to 13 multiply-accumulates per cycle for selected algorithm kernels. Finally, we combined all of our new mechanisms and threading models to create a multigranular parallel version of equake which is faster than either threading model alone.

References

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

[2] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 291, 2003.

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

[4] Arvind, "Dataflow: Passing the token." ISCA Keynote, June 2005.

[5] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek, "Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine," in *Proceedings of the4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[6] 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*, vol. 13, pp. 451–490, October 1991.

[7] 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*, (Palo Alto, California), pp. 210–215, IEEE Computer Society and ACM SIGARCH, April 3–5, 1978.

[8] 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*, pp. 226–234, IEEE Computer Society Press, 1986.

[9] J. R. Gurd, C. C. Kirkham, and I. Watson, "The manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[10] M. Kishi, H. Yasuhara, and Y. Kawamura, "Dddp-a distributed data driven processor," in *Conference Proceedings of the tenth annual international symposium on Computer architecture*, pp. 236–242, IEEE Computer Society Press, 1983.

[11] 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*, pp. 36–45, ACM Press, 1989.

[12] G. M. Papadopoulos and D. E. Culler, "Monsoon: an explicit token-store architecture," in *Proceedings of the 17th annual international symposium on Computer Architecture*, pp. 82–91, ACM Press, 1990.

[13] 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*, (Portland, Oregon), pp. 45–54, IEEE Computer Society TC-MICRO and ACM SIGMICRO, December 1–4, 1992. SIG MICRO Newsletter 23(1–2), December 1992.

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

[15] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial computation," *SIGPLAN Not.*, vol. 39, no. 11, pp. 14–26, 2004.

[16] S. Swanson, M. Mercaldi, A. Putnam, A. Petersen, A. Schwerin, M. Oskin, and S. Eggers, "Balancing parallelism and sequentiality in dataflow models: Wave-ordered memory," Tech. Rep. UWCSE-2005-10-3, UW-Computer Science and Engineering, 2005.

[17] 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.

[18] 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.

[19] W. Lee *et al.*, "Space-time scheduling of instruction-level parallelism on a Raw machine," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS-VIII*, October 1998.

[20] 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*, pp. 161–171, ACM Press, 2000.

[21] S. C. Goldstein and M. Budiu, "Nanofabrics:spatial computing using molecular electronics," in *Proceedings of the 28th annual international symposium on Computer architecture*, pp. 178–191, 2001.

[22] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Trans. Comput.*, vol. 36, no. 5, pp. 547–553, 1987.

[23] "A performance model to guide instruction scheduling on spatial computers." In submission to CGO 2006.

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

[25] A. Putnam, S. Swanson, M. Mercaldi, K. Michelson, A. Petersen, A. Schwerin, M. Oskin, and S. Eggers, "The microarchitecture of a pipelined wavescalar processor: An RTL-based study," Tech. Rep. UWCSE-2005-10-2, UW-Computer Science and Engineering, 2005.

[26] "Silicon design chain cooperation enables nanometer chip design." Cadence Whitepaper. http://www.cadence.com/whitepapers/.

[27] "Cadence website." http://www.cadence.com.

[28] 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*, pp. 14–24, IEEE Computer Society, 2002.

[29] 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 *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), p. 29, IEEE Computer Society, 2003.

[30] R. Desikan, D. Burger, S. Keckler, and T. Austin, "Sim-alpha: a validated, execution-driven alpha 21264 simulator," Tech. Rep. TR-01-23, UT-Austin Computer Sciences, 2001.

[31] A. J. et. al., "A 1.2ghz alpha microprocessor with 44.8gb/s chip pin bandwidth," in *IEEE International Solid-State Circuits Conference*, vol. 1, pp. 240–241, 2001.

[32] K. Krewel, "Alpha ev7 processor: A high-performance tradition continues," *Microprocessor Report*, April 2005.

[33] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: a tutorial," *IEEE Computer*, vol. 29, pp. 66–76, Dec. 1996.

[34] J. R. Goodman, M. K. Vernon, and P. J. Woest, "Efficent synchronization primitives for large-scale cache-coherent multiprocessors," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, (Boston, Massachusetts), pp. 64–75, 1989.

[35] D. Tullsen, J. Lo, S. Eggers, and H. Levy, "Supporting fine-grain synchronization on a simultaneous multithreaded processor," in *Proceedings of the 5th InternationalSymposium on High Performance Computer Architecture*, 1999.

[36] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee, "Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor," in *ISCA*, pp. 306–317, 1998.

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

[38] T. Shimada, K. Hiraki, and K. Nishida, "An architecture of a data flow machien and its evaluation," in *Digest of Papers, COMPCON Spring 84*, pp. 486–490, IEEE, 1984.

[39] 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*, (Toronto, Ontario), pp. 342–351, ACM SIGARCH and IEEE Computer Society TCCA, May 27–30, 1991. *Computer Architecture News,* 19(3), May 1991.

[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 Trans. Comput. Syst.*, vol. 15, no. 3, pp. 322–354, 1997.

[41] M. Ekman and P. Stenström, "Performance and power impact of issue-widt h in chip-multiprocessor cores," in *International Conference on Paralllel Processing*, 2003.

[42] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukolun, "The stanford hydra CMP," *IEEE Micro*, vol. 20, march/april 2000.

[43] 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*, (Vancouver, British Columbia), pp. 282–293, IEEE Computer Society and ACM SIGARCH, June 12–14, 2000. *Computer Architecture News,* 28(2), May 2000.

[44] SPEC, "Spec CPU 2000 benchmark specifications." SPEC2000 Benchmark Release, 2000.