# The WaveScalar Architecture

STEVEN SWANSON, ANDREW SCHWERIN, MARTHA MERCALDI, ANDREW PETERSEN, ANDREW PUTNAM, KEN MICHELSON, MARK OSKIN, and SUSAN J. EGGERS
University of Washington

Silicon technology will continue to provide an exponential increase in the availability of raw transistors. Effectively translating this resource into application performance, however, is an open challenge that conventional superscalar designs will not be able to meet. We present WaveScalar as a scalable alternative to conventional designs. WaveScalar is a dataflow instruction set and execution model designed for scalable, low-complexity/high-performance processors. Unlike previous dataflow machines, WaveScalar can efficiently provide the sequential memory semantics that 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 ones 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 article 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.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Data-flow architectures*; C.5.0 [**Computer System Implementation**]: General; D.4.1 [**Operating Systems**]: Process Management—*Threads, concurrency*

General Terms: Performance, Experimentation, Design

Additional Key Words and Phrases: WaveScalar, dataflow computing, multithreading

Authors' address: S. Swanson (contact author), A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers, Department of Computer Science and Engineering, Allen Center for CSE, University of Washington, Box 352350, Seattle, WA 98195; email: swanson@cs.ucsd.edu.

**ACM Reference Format:**
Swanson, S., Schwerin, A., Mercaldi, M., Petersen, A., Putnam, A., Michelson, K., Oskin, M., and Eggers, S. J. 2007. The WaveScalar architecture. ACM Trans. Comput. Syst. 25, 2, Article 4 (May 2007), 54 pages. DOI = 10.1145/1233307.1233308 http://doi.acm.org/10.1145/1233307.1233308

## 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 [Agarwal et al. 2000]. 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, specifically, 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 [Swanson et al. 2003], that addresses these challenges by building on the dataflow execution model [Dennis and Misunas 1975]. 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 upon which the von Neumann model relies, 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 [Arvind 2005] 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 needed by imperative languages.

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 shows 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 this 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 article 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 demonstrates that this WaveCache performs comparably to a modern out-of-order superscalar design, but requires 20% less silicon area.

Next, we extend WaveScalar and the WaveCache to support conventional pthread-style threading. The changes to WaveScalar include lightweight 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 with conventional multithreading. We describe WaveScalar's "unordered" memory interface and show how it can be used 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 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 article 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 we can combine both threading styles to enhance performance. Finally, Section 7 concludes.

## 2. SINGLE-THREADED WAVESCALAR

Although the dataflow model that WaveScalar uses is fundamentally different than the von Neumann model that dominates conventional designs, 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 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 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 that these programs require.

To provide context for our description, we first describe how the von Neumann model accomplishes the tasks previously outlined 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 presented 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 in which 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 these 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 that 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, which 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 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 [Swanson et al. 2003] (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.

Next 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

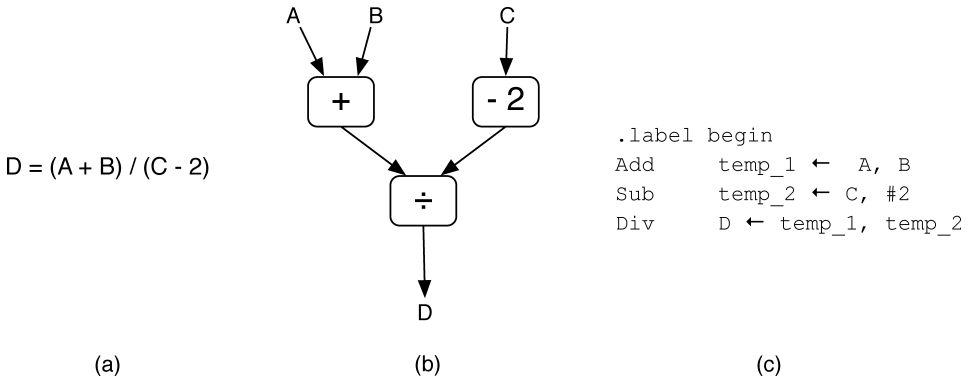


Fig. 1. A simple dataflow fragment: (a) a simple program statement; (b) its dataflow graph; and (c) the corresponding WaveScalar assembly. The order of the WaveScalar assembly statements is unimportant, since they will be executed in dataflow fashion.

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 pseudoregisters in a compiler's program representation, there can be an arbitrary number of them. Second, the order of 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 lefthand 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

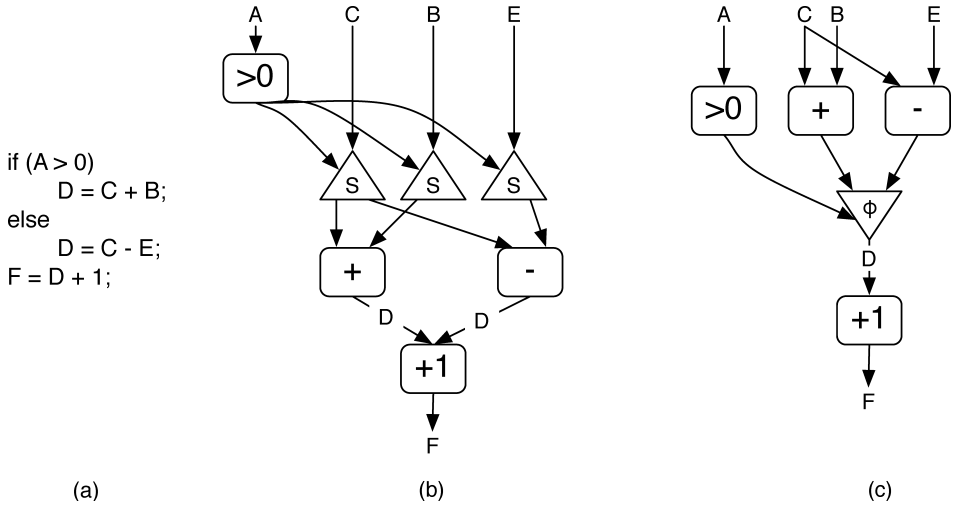


Fig. 2. Implementing control in WaveScalar: (a) an IF-THEN-ELSE construct and equivalent dataflow representations; (b) STEER instructions (triangles labeled "s") ensure that only one side of the branch executes, while (c) computes both sides and a $\phi$ instruction selects the result to use.

through the executable so that 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 a source of some overhead in the form of extra static instructions.

WaveScalar uses STEER instructions to steer values to the correct path and $\phi$ instructions for predication. The STEER [Culler et al. 1991] 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 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, namely a predicate and two operands, and steers the result depending on the predicate. WaveScalar provides a steering version for all one- and two-input instructions.

The $\phi$ instruction [Cytron et al. 1991] takes two input values and a Boolean selector input and, depending on the selector, passes one of the inputs to its output. Moreover, $\phi$ 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 $\phi$ instructions in action.

2.2.3 *Loops and Waves.* The STEER instruction may appear to be sufficient for WaveScalar to express loops, since it provides a basic branching facility.

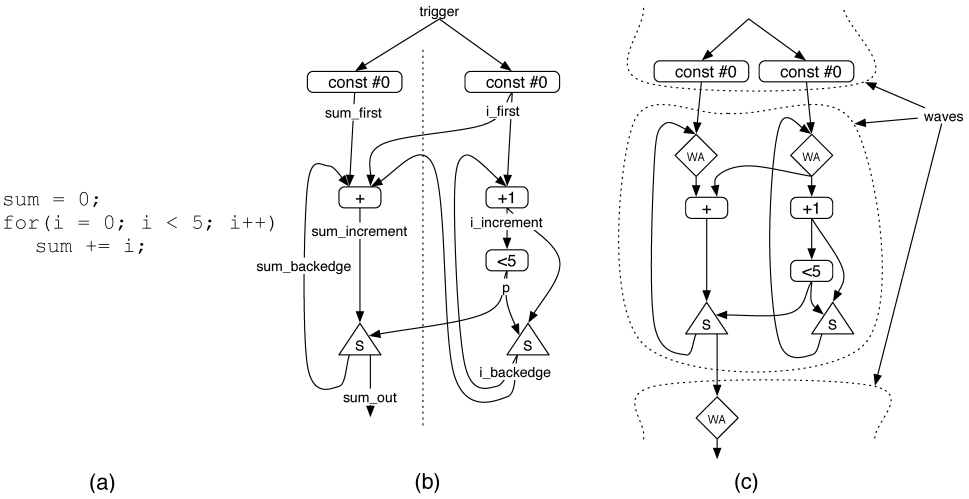


Fig. 3. Loops in WaveScalar: (a) a simple loop; (b) a naive, slightly broken dataflow implementation; and (c) the correct WaveScalar implementation.

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 illustrates both 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 five 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 righthand-side STEER instruction produces no output.

The problem arises because the dataflow execution model makes no guarantee about how long it takes for 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, 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 nor wise.

Previous dataflow machines provided one of two solutions. In the first, *static dataflow* [Dennis and Misunas 1975; Davis 1978], 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* [Shimada et al. 1986; Gurd et al. 1985; Kishi et al. 1983; Grafe et al. 1989; Papadopoulos and Culler 1990], 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 of 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* [Swanson et al. 2003]. 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 did most dynamic dataflow machines), WaveScalar assigns them to compiler-delineated portions of the dataflow graph, called *waves*. Waves are similar to hyperblocks [Mahlke et al. 1992], but are more general, since they can both contain control-flow joins and have more than one entrance. They cannot contain loops. Figure 3(c) shows the example loop divided into waves (as shown by dotted lines). At the top of each wave is a set of WAVE-ADVANCE instructions (small diamonds), each of which increments the wave number of the value that passes through it.

Assume that 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 righthand-side STEER instruction will produce 1.1 and pass it back to 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 five iterations, the left STEER instruction produces the final value of `sum`: 5.10, which flows directly into 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 operand tags match. More generally, wave 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 (see Section 2.2.5).

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

[1]The execution model does not specify where the data values are stored nor how matching takes place. Efficiently storing and matching input tokens is a key challenge in dynamic dataflow architecture, and Section 3 discusses this.

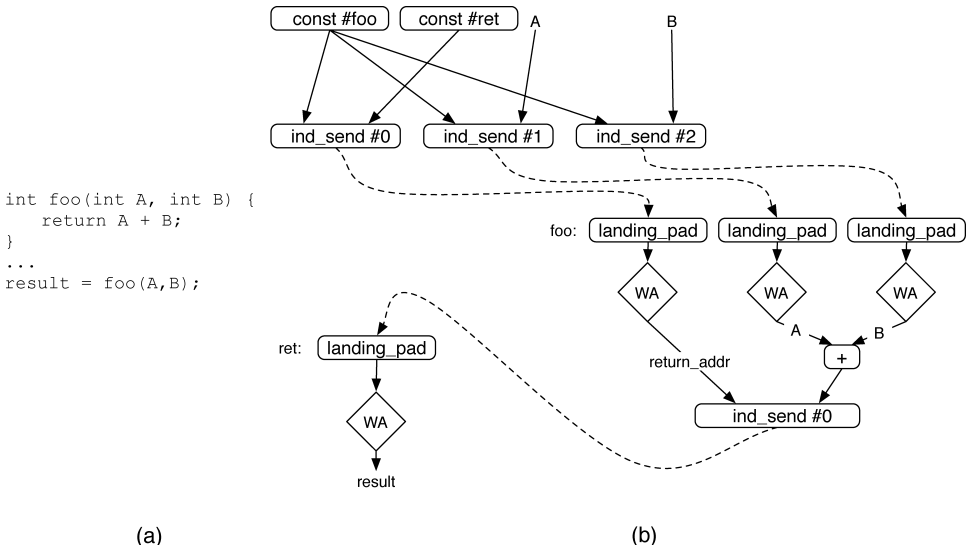


Fig. 4. A function call: (b) the dataflow graph for (a) a call to a simple function. The lefthand-side of the dataflow graph uses INDIRECT-SEND instructions to call function `foo` on the right. The dashed lines show data dependences that WaveScalar must resolve at runtime. The immediate values on the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

function and trigger its execution. Passing arguments creates a data dependence between the caller and 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.

Figure 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 runtime. 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). Another INDIRECT-SEND is used to return from the function.

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 language memory semantics is one of the key challenges that has prevented dataflow processing from

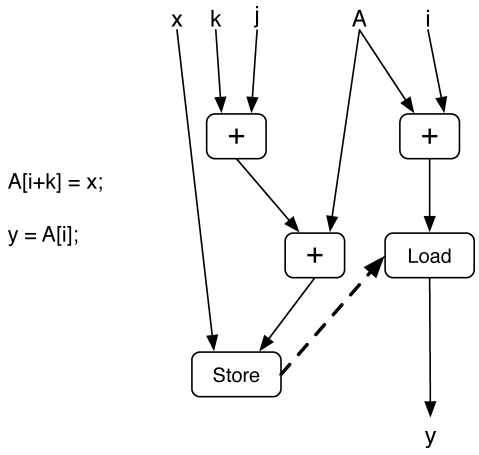


Fig. 5. Program order. The dashed line represents an implicit potential data dependence between the store and load instructions that conventional dataflow instruction sets have difficulty expressing. Without the dependence, the dataflow graph provides no ordering relationship between the memory operations.

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 ensuring 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 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. Firstly, they must ensure that the memory operations within a wave execute in the correct order. Wave-ordered memory achieves this by giving each memory operation in a wave a *sequence number*. Sequence numbers increase on all paths through a wave, ensuring that if one memory operation has a larger

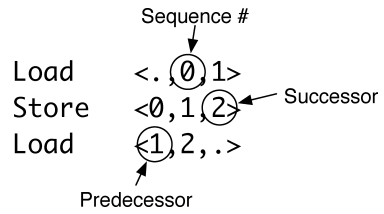


Fig. 6. Simple wave-ordered annotations. The three memory operations must execute in the order shown. Predecessor, sequence, and successor numbers encode the ordering constraints. The "." symbols indicate that operations 0 and 2 are the first and last operations, respectively, in the wave.

sequence number than another, the one with the larger number comes later in the 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.

Secondly, 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 all memory operations with lower sequence numbers to complete. Control flow complicates this method because it allows some of the memory operations to execute (those on taken paths) while others do not (those on the nontaken paths). To accommodate, wave-ordered memory must distinguish between operations that take a long time to fire and those that never will. To ensure that all memory operations on the correct path are executed, each memory operation also carries the sequence number of its previous and subsequent operations in the 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 a 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 lefthand portion of Figure 7 shows a simple IF-THEN-ELSE control flow graph that demonstrates how the wildcard is applied; the righthand portion depicts how memory operations on the taken path are sequenced, described next.

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

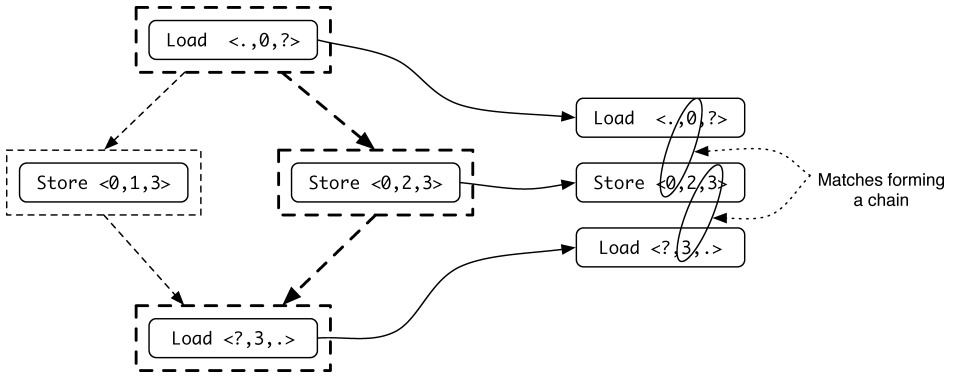


Fig. 7. Wave-ordering and control. Dashed boxes and lines denote basic blocks and control paths. The righthand-side of the figure shows the instructions that actually execute when control takes the righthand path (bold lines and boxes) and the matches between their annotations that define program order.

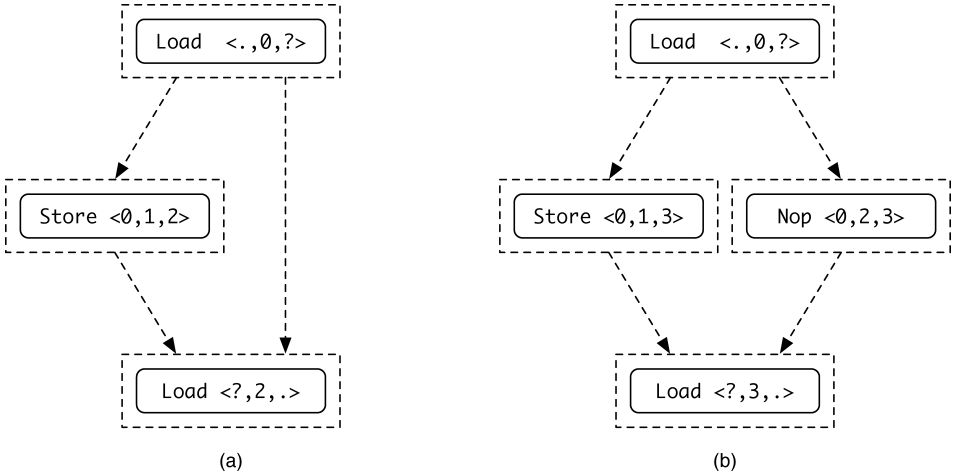


Fig. 8. Resolving ambiguity. In (a) chaining memory operations is impossible along the right-side path; (b) the addition of a MEMORY-NOP allows chaining.

wave. The chain must begin with an operation whose sequence number is 0 and end with successor number ".", indicating that there is no successor.

It is easy to enforce this condition on the beginning and end of the chain of operations, but ensuring that all possible paths through the wave are complete is more difficult. Figure 8(a) shows an example. The branch and join mean that instruction 0's successor and instruction 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 righthand path. To create a chain, the compiler inserts a special MEMORY-NOP instruction between 0 and 2 on the righthand 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% of static instructions.

—*Ordering rules.* We can now demonstrate how WaveScalar uses wave numbers and the aforementioned annotations to construct a total ordering over all memory operations in a program. Figure 7 shows a simple example. Control takes the righthand path, resulting in three memory operations executed. At the right, ovals show the links between the three operations that form them into a chain. The general rule is that a link exists 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 global interwave ordering with local intrawave ordering provides a total ordering on all operations in the program.

—*Expressing parallelism.* The basic version of wave-ordered memory described earlier 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. Note that the predecessor numbers are still necessary to prevent a store from executing before the preceeding loads have completed.

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 ovals 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 either order or 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 lefthand branch, loads 3 and 7 can execute as soon as store 2 has executed.

Wave-ordered memory can express parallelism among load and store operations that a conventional out-of-order processor would discover by speculatively assuming memory independence [Chrysos and Emer 1998]. The speculative

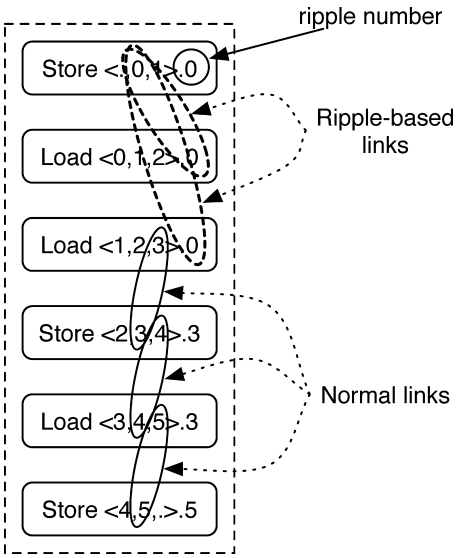


Fig. 9. Simple ripples. A single wave containing a single basic block. The ripple annotations allow loads 1 and 2 to execute in either order or in parallel, while the stores must wait for all previous loads and stores to complete. Ovals depict the links formed between operations.

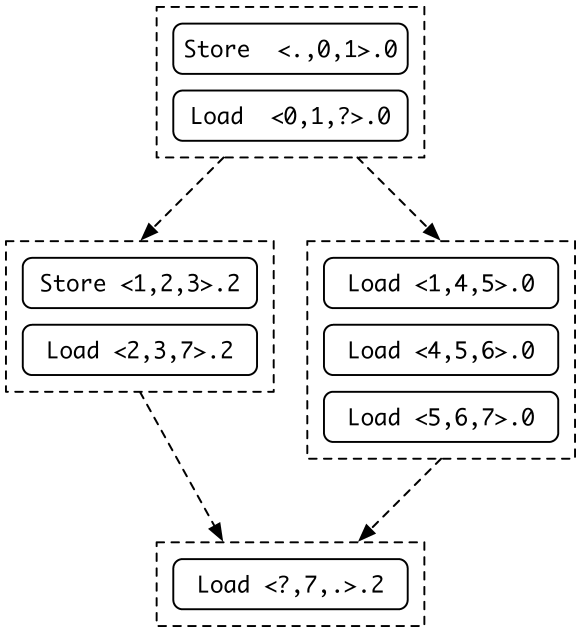


Fig. 10. Ripples and control. Branches make ripple behavior more complicated. If control takes the righthand path, Loads 1 and 4–6 can execute in any order, but Load 7 must wait for an operation with a sequence number greater than 2.

approach can also uncover some paralellism that wave-ordered memory cannot express (e.g., the compiler may be unable to prove that two stores are independent when they actually are). However, nothing in the WaveScalar instruction set or execution model prevents a WaveScalar processor from speculatively issuing memory operations and using the wave-ordering information to catch and correct mispeculations. Our implementation does not currently speculate.

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 [Beck et al. 1991; Budiu et al. 2004]. 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 [Swanson 2006].

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).

Previous dataflow machines have also provided two memory structures, I-structures and M-structures, intended to support functional programming languages. These structures combine memory ordering with synchronization.

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

Dataflow languages such Id [Nikhil 1990] and SISAL [Feo et al. 1995] provide this ability with I-structures [Arvind et al. 1989]. I-structures are write-once memory structures. When a program allocates an I-structure, it is *empty* and contains no value. A program can write, or fill-in, an I-structure (at most) once. Reading from an empty I-structure blocks until the I-structure is full. Reading from a full I-structure returns the value it holds. In the array example given before, one thread allocates an array of I-structures and starts filling them in. The second thread can attempt to read entries of the array, but will block if it tries to access an empty I-structure.

—*M-structures.* M-structures [Barth et al. 1991] provide checkin/checkout semantics for variables. Reading from a full M-structure removes the value, and a write fills the value back in. Attempting to read from an empty M-structure blocks until the value is returned.

A typical example of M-structures in action is a histogram. Each bucket is an M-structure, and a group of threads adds elements to the buckets concurrently.

Since addition is commutative, the order of updates is irrelevant, but they must be sequentialized. M-structures provide precisely the necessary semantics.

### 2.3 Discussion

The WaveScalar instruction set that 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 dataflow ISA in hand, our task is to develop a decentralized, scalable architecture to match. In addition to the scaling challenges, this architecture also must address additional challenges specific to WaveScalar. In particular, it 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 does 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, that of defect- and fault-tolerance, is the subject of ongoing research. The WaveCache's

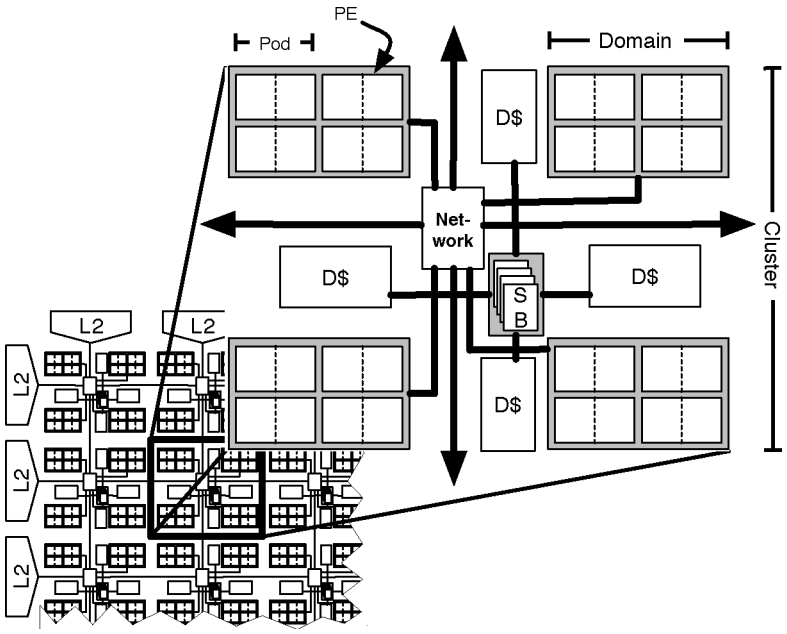


Fig. 11. The WaveCache. The hierarchical organization of the microarchitecture of the WaveCache.

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. Finally, Section 3.8 describes other processor designs that share many of WaveScalar's goals. Section 4 evaluates the design in terms of performance and the amount of area it requires.

### 3.1 WaveCache Architecture Overview

Several recently proposed architectures, including WaveCache, take a tile-based approach to addressing the scaling problems outlined in the Introduction [Nagarajan et al. 2001; Sankaralingam et al. 2003; Lee et al. 1998; Mai et al. 2000; Goldstein and Budiu 2001; Budiu et al. 2004]. 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 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* (see Figure 11). A cluster contains four identical *domains*, each with eight identical processing elements

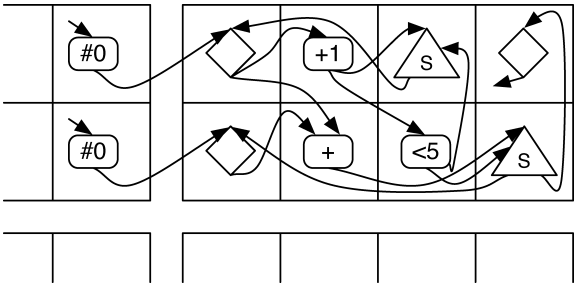


Fig. 12. Mapping instruction into the WaveCache. The loop in Figure 3(c) mapped onto two WaveCache domains. Each large square is a processing element.

(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 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) so as 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 (i.e., everything but the CONST instructions that initiates the loop) has been placed in a single domain.

A processing element's chief responsibilities are 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 the large, centralized associative tag-matching store found on some previous dataflow machines [Gurd et al. 1985]. 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 other's 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 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 MESI protocol 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 found 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 has five stages (see Figure 13):

(1) *Input*. At the INPUT stage, operand messages arrive at the PE either from itself or another PE. The PE may reject messages if more than three arrive in one cycle; the senders then retry on a later cycle.
(2) *Match*. During MATCH, operands enter the *matching table*. The matching table contains a tracker board and operand caches. It determines which instructions are ready to fire and issues eligible instructions by placing their matching table index into the instruction scheduling queue.
(3) *Dispatch*. At the DISPATCH stage, 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, this stage speculatively issues the consumer instruction to the scheduling queue.
(4) *Execute*. The EXECUTE stage executes an instruction. Its result goes to the output queue and/or the local bypass network.

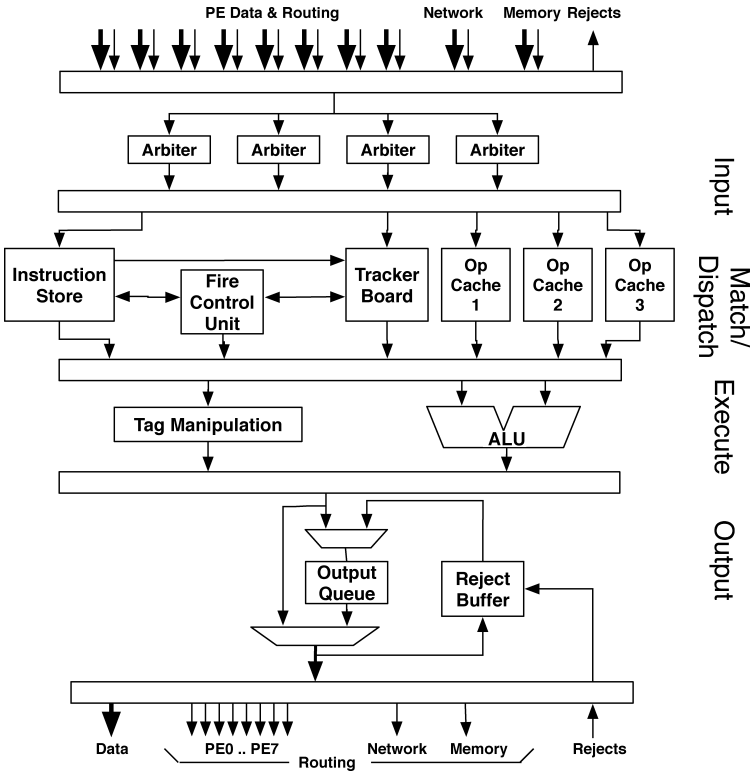


Fig. 13. PE block diagram. The processing element's structure by pipeline stage. Note that the block at the end of output is the same as the block at the start of input, since wire delay is spread between the two stages.

(5) *Output*. An instruction output is sent to its consumer instructions via the intradomain network during the OUTPUT stage. Consumers may be at this PE or a remote PE.

An instruction store holds the decoded instructions that reside at a PE. To keep it single-ported, the RTL design divides it into several small SRAMs, each holding the decoded information needed at a particular stage of the pipeline. The instruction store comprises about 33% of the PE's area.

The matching table handles instruction input matching. Implementing this operation cost-effectively is essential to an efficient dataflow machine. The key challenge in designing WaveScalar's matching table is emulating a potentially infinite table with a much smaller physical structure. This problem arises because WaveScalar is a dynamic dataflow architecture with no limit on the number of dynamic instances of a static instruction with unconsumed inputs. We use a common dataflow technique [Gurd et al. 1985; Shimada et al. 1986] to address this challenge: The matching table is a specialized cache for a larger, in-memory matching table. New tokens are stored in the matching cache. If a token resides there for a sufficiently long time, another token may arrive that hashes to the same location. In this case, the older token is sent to the matching table in memory.

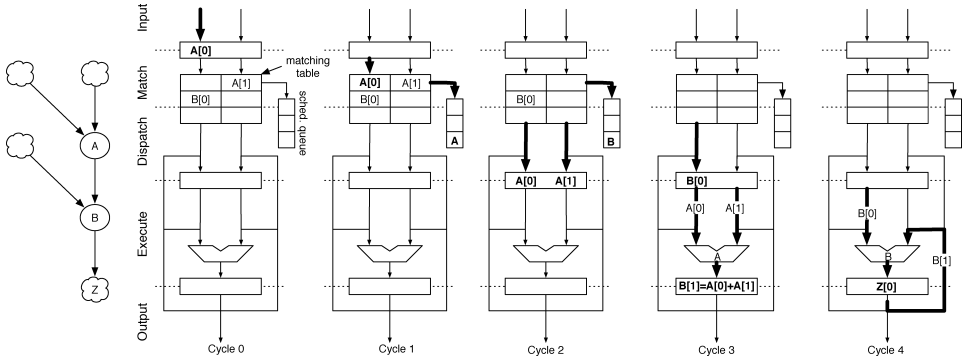


Fig. 14. The flow of operands through the PE pipeline and forwarding networks.

The matching table is separated into three columns, one for each potential instruction input (certain WaveScalar instructions, such as data steering instructions, can have three inputs.)[2] Each column is divided into four banks to allow up to four messages to arrive at each cycle. Reducing the number of banks to two reduced performance by 5% on average and 15% for *ammp*. Increasing the number of banks to eight had negligible effect. In addition to the three columns, the matching table contains a *tracker board* which holds operand tags (wave number and consumer instruction number) and tracks which operands are present in each row of the matching table.

Since the matching table is a cache, we can apply traditional cache optimizations to reduce its miss rate. Our simulations show that two-way set-associativity increases performance over direct-mapped by 10% on average and reduces matching table misses (situations when no row is available for an incoming operand) by 41%. Four-way associativity provides less than 1% additional performance, hence the matching table is two-way. The matching table comprises about 60% of PE area.

To achieve good performance, PEs must be able to execute dependent instructions on consecutive cycles. When DISPATCH issues an instruction with a local consumer of its result, it speculatively schedules the consumer instruction to execute on the next cycle. The schedule is speculative because DISPATCH cannot be certain that the dependent instruction's other inputs are available. If they are not, the speculatively scheduled consumer is ignored.

Figure 14 illustrates how instructions from a simple dataflow graph (on the lefthand-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 $n$th input to instruction $X$. Five consecutive cycles are depicted; before the first of these, one input for each of instructions $A$ and $B$ has arrived and resides in the matching table, and the corresponding bits are also set in the tracker board. The tracker board also contains the tag (THREAD-ID and WAVE-NUMBER) of the values in each occupied row. The "clouds"

[2]The third column is special and supports only single-bit operands. This is because three-input instructions in WaveScalar always have one argument which needs to be only a single bit. Other columns hold full 64-bit operands.

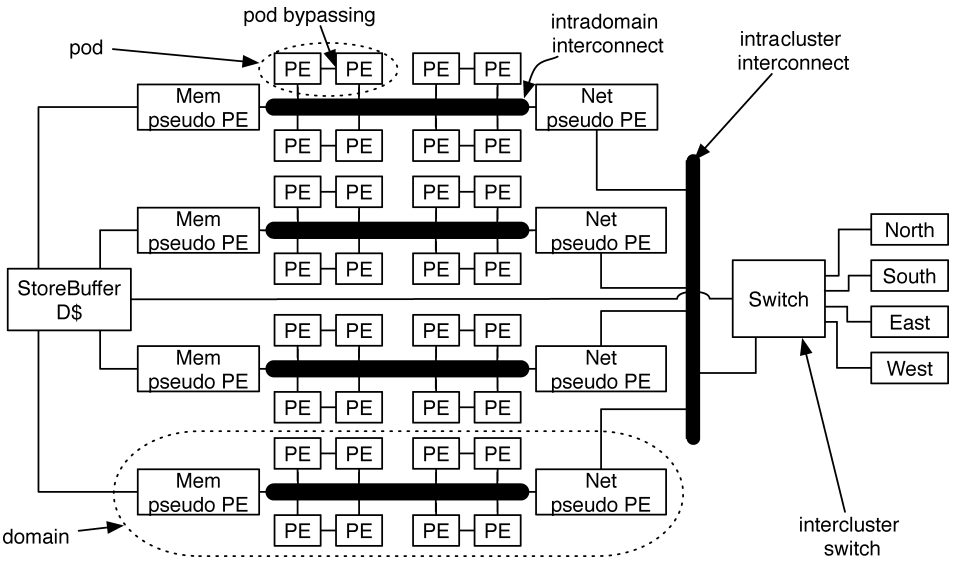


Fig. 15. The cluster interconnects. A high-level picture of a cluster illustrating the interconnect organization.

in the dataflow graph represent operands that were computed by instructions at other processing elements and have arrived via the input network.
—*Cycle* 0: Operand $A[0]$ arrives and INPUT accepts it (at left in Figure 14).
—*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 logic that executes back-to-back dependent instructions within our cycle-time goal.

### 3.3 The WaveCache Interconnect

The previous section described the execution resource of the WaveCache, namely, the PE. This section will detail how PEs on the same chip communicate. PEs send and receive data using a hierarchical on-chip interconnect (see Figure 15). There are four levels in this hierarchy: intrapod, intradomain, intracluster, and intercluster. While the purpose of each network is the same, that is, 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 intrapod 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 that obtained by using the intradomain interconnect (described next). Although PEs in a pod snoop each other's bypass networks, the rest of their hardware remains partitioned, that is, 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 intradomain interconnect 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. However, their tightly integrated design added significant complexity and took a great deal of effort to implement correctly. Integrating more PEs would increase complexity further, and our data showed that the additional gains in performance would be small.

3.3.2 *The Intradomain Interconnect.* PEs communicate with PEs in other pods over an intradomain interconnect. In addition to the eight PEs in the domain, the intradomain interconnect also connects two *pseudo-PEs* that serve as gateways to the memory system (the MEM pseudo-PE) and to other PEs on the chip (the NET pseudo-PE). The pseudo-PEs' interface to the intradomain network is identical to a normal PE's.

The intradomain 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 intradomain network using an ACK/NACK network.

3.3.3 *The Intracluster Interconnect.* The intracluster interconnect provides communication between the four domains' NET pseudo-PEs. It also uses an ACK/NACK network similar to that of the intra-domain interconnect.

3.3.4 *The Intercluster Interconnect.* The intercluster 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 intercluster network switch which routes messages between six input/output ports: four of the ports lead to network switches in the four cardinal directions, one is shared among the four domains' NET pseudo-PEs, and one is dedicated to the store buffer and L1 data cache.

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

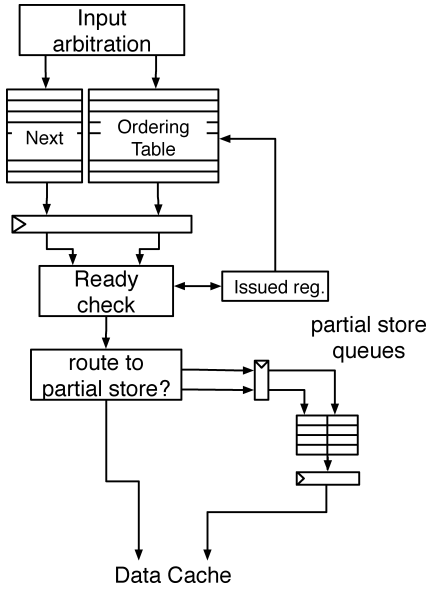


Fig. 16. The microarchitecture of the store buffer.

to have space. The intercluster switch provides two virtual channels that the interconnect uses to prevent deadlock [Dally and Seitz 1987].

### 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 intercluster interconnect. All memory requests for a single *dynamic* instance of a wave (e.g., an iteration of an inner loop), including requests from both local and remote processing elements, are managed by the same store buffer.

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

The store buffer (see Figure 16) contains four major microarchitectural components: an *ordering table*, a *next-request 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-request tables. MEMORY-SCHEDULE reads up to four requests (one from each of the four banks) 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 to follow). We detail each pipeline stage of this memory interface to next.

MEMORY-INPUT accepts up to four new memory requests per cycle. It writes the address, operation, and data (if available in the case of stores) into the ordering table at the index seq($R$). If succ($R$) is defined (i.e., not "?"), the entry in the next-request table at location seq($R$) is updated to succ($R$). If pred($R$) is defined, the entry in the next-request table at location pred($R$) is set to seq($R$).

MEMORY-SCHEDULE maintains the issued register which points to the subsequent memory operations to be dispatched to the data cache. It uses this register to read four entries from the next-request and ordering tables. If any memory ordering links can be formed (i.e., next-request 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 next. 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 operations to other addresses may proceed. When their data finally arrives, all operations in the partial-store queue can be applied in quick succession. Each WaveScalar 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 present. The memory 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 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 its partial-store queue is full. The memory operation remains in the ordering table 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 that is distributed along the edge of the chip (16 banks in a 4x4 WaveCache). A directory-based MESI coherence protocol keeps the L1 caches consistent. All coherence traffic travels over the intercluster interconnect.

The L1 data cache has a three-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 investigation, the tradeoffs involved, and a placement's effects on performance are available in Mercaldi et al. [2006a, 2006b]. Here, we describe the approach we used for the studies in this article.

The placement scheme has a compile-time and a runtime component. The compiler is responsible for grouping instructions into segments. At runtime, a whole segment of instructions will be placed at the same PE. Because of this, the compiler tries to group instructions into the same segment if these instructions are not likely to execute simultaneously but share operands, and therefore can utilize the fast local bypass network available inside of each PE. The algorithm we use to form segments is a depth-first traversal of the dataflow graph.

At runtime, the WaveCache loads a segment of instructions when an instruction that is not mapped into the WaveCache needs to execute. As previously discussed, the entire segment is mapped to a single PE. Because of the ordering that 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 even rows and right-to-left on odd rows.

This placement scheme does a good job of scheduling for minimal execution resource contention and communication latency. However, a third factor, the so-called "parallelism explosion," can also have a strong effect on performance in dataflow systems. The parallelism explosion occurs when part of an application (e.g., the index computation of an inner loop) runs ahead of the rest of the 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 [Culler 1990], to restrict the number iterations $k$ that can be executing at one time. We tune $k$ for each application, and for the applications we study, it is between two and five.

### 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 earlier. We use the RTL model, combined with detailed architectural simulation, to tune the WaveCache's parameters and make tradeoffs between performance, cycle time, and silicon area. All the specific parameters of the architecture (e.g., cache sizes, bus widths, etc.) 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 the choice of particular parameters,

Table I. A Cluster's Area Budget

| Component | Fraction of Cluster Area |
|---|---|
| PE stages | |
| INPUT | 0.9% |
| MATCH | 43.3% |
| DISPATCH | 0.4% |
| EXECUTE | 1.8% |
| OUTPUT | 1.3% |
| instruction store | 23.2% |
| PE total | 71% |
| Domain overhead | 7.4% |
| Intercluster interconnect switch | 0.9% |
| Storebuffer | 6.2% |
| L1 cache | 14.5% |

A breakdown of the area required for a cluster. Most of the area is devoted to processing resources.

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 article (but can be found in Swanson et al. [2006]). Here, we summarize the methodology and timing results.

We derive our results with the design rules and recommended tool infrastructure of the Taiwan Semiconductor Manufacturing Company's TSMC Reference Flow 4.0 [TSMC 2007], 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 [Cadence 2007]. 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 (∼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 I shows a breakdown of area within a single cluster. The ratios for an array of clusters are the same.

In the next section, we place the WaveCache in context relative to other tiled architectures. Then, in Section 4 we evaluate WaveCache's performance on single-threaded applications and compare this as well as its area requirements with a conventional superscalar processor.

### 3.8 Other Tiled Architectures

The WaveCache hardware design described in Sections 3 and 5 is a tiled architecture. Broadly speaking, a tiled architecture is a processor design that uses an array of basic building blocks of silicon to construct a larger processor.

Tiled architectures provide three advantages over traditional monolithic designs. First, they reduce design complexity by emphasizing design reuse.

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

Several research groups have proposed tiled architectures with widely varying tile designs. Smart Memories [Mai et al. 2000] provides multiple types of tiles (e.g., processing elements and reconfigurable memory elements). This approach allows greater freedom in configuring an entire processor, since the mix of tiles can vary from one instantiation to the next, perhaps avoiding the difficulties in naive scaling that we found in our study.

The TRIPS [Nagarajan et al. 2001; Sankaralingam et al. 2003] processor uses dataflow ideas to build a hybrid von Neumann/dataflow machine. It uses a program counter to guide execution, but instead of moving from one instruction to the next, the TRIPS PC selects *frames* (similar to hyperblocks [Mahlke et al. 1992]) of instructions for execution in an array of 16 processing elements that make up a TRIPS processor.

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

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

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

WaveScalar and TRIPS also take similar approaches to ordering memory operations. TRIPS uses load/store IDs (LSIDs) [Smith et al. 2006] to order memory

Table II. Microarchitectural Parameters of the Baseline WaveCache

| | |
|---|---|
| WaveCache Capacity | 2K(WC1×1) or 8K(WC2×2) static instructions (64/PE) |
| PEs per Domain | 8 (4 pods); 1 FPU/domain |
| PE Input Queue | 16 entries, 4 banks (1KB total); 32 cycle miss penalty |
| PE Output Queue | 4 entries, 2 ports (1r, 1w) |
| PE Pipeline Depth | 5 stages |
| Domains/Cluster | 4 |
| Network Switch | 2-port, bidirectional |
| Network Latency | within Pod: 1 cycle<br>within Domain: 5 cycles<br>within Cluster: 9 cycles<br>Intercluster: 9 + cluster dist. |
| L1 Caches | 32KB, 4-way set-associative, 128B line,<br>4 accesses per cycle |
| L2 Cache | 1MB (WC1×1) or 4MB (WC2×2) shared, 128B line,<br>16-way set-associative, 10 cycle access |
| Main RAM | 200 cycle latency |

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

The RAW project [Taylor et al. 2004] uses a simple processor core as a tile and builds a tightly coupled multiprocessor. The RAW processor provides for several different execution models. The compiler can statically schedule a single program to run across all the tiles, effectively turning RAW into a VLIW-style processor. Alternatively, the cores can run threads from a larger computation that communicates using RAW's tightly integrated, interprocessor message-passing mechanism.

## 4. SINGLE-THREADED WAVECACHE PERFORMANCE

This section measures 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 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 II 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

Table III. Workload Configurations

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

Workloads and parameters used in this article.

report here is lower than that in the original WaveScalar paper [Swanson et al. 2003]. 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*, and *twolf*; from SpecFP2000: *ammp*, *art*, and *equake* [SPEC 2000]; and from Mediabench[Lee et al. 1997]: *djpeg*, *mpeg2encode*, and *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 nor its execution model. We are currently working on a full-fledged compiler that will allow us to run a wider range of applications [Petersen et al. 2006]. Table III shows the configuration parameters for these workloads, as well as the multithreaded workloads we use in Section 5. We skip past the initialization phases of all our workloads.

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 [Hrishikesh et al. 2002]. AIPC measures the number of nonoverhead instructions (e.g., STEER, $\phi$, etc.) executed per cycle. The AIPC measurements for the superscalar architectures to which we compare WaveScalar are in good agreement with other measurements [Mukherjee et al. 2003].

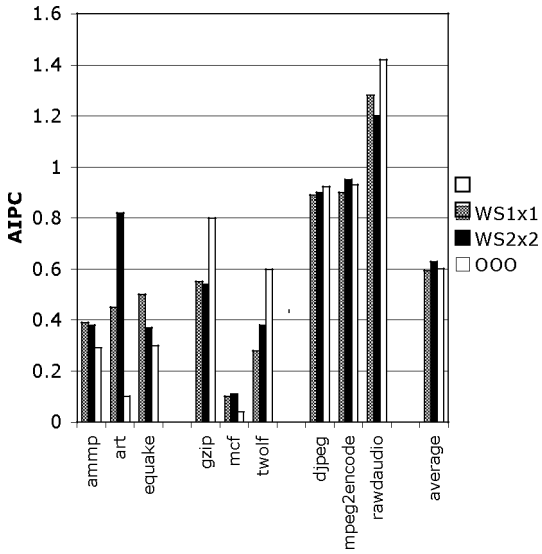


Fig. 17. Single-threaded WaveCache vs. superscalar. On average, both WaveCaches perform comparably to the superscalar.

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 [Desikan et al. 2001; Jain 2001], but with the same L1, L2, and main memory latencies that we model for the WaveCache.[3] The two WaveCache configurations are *WC1x1*, a 1×1 array of clusters, and *WC2x2*, a 2×2 array. The only other difference between the two architectures is the size of the L2 cache (1MB for WC1x1 versus 4MB for WC2x2).

Figure 17 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 four-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, avoiding this overhead. Second, the average latency of messages between instructions increases by 20% on average because some messages must traverse the intercluster network. The other applications, except for *twolf* and *art*, have very similar performance on both configurations. *Twolf* and *art* do better on

[3]Note that the load-use penalty for the WaveCache is still longer: It takes one to two cycles for the request to travel through the wave-ordering store buffer and reach the L1 cache.

WC2x2; their working sets are large enough to utilize either 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 execute 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 most 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 that the compiler can create by 50% on average compared to floating point applications, consequently reducing memory parallelism. *Twolf* and *gzip* are hit hardest by this effect, and OOO outperforms 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 further clarified 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 [Jain 2001; Krewel 2005]. 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 interchip network controller), but which would be present in a real WaveCache. We estimate the remaining area to be ∼291mm$^2$, with ∼160mm$^2$ devoted to 2MB of L2 cache. Scaling all these measurements to 90nm technology yields ∼72mm$^2$ total and 40mm$^2$ of L2. Measurements from our RTL model show that WC1x1 occupies 48mm$^2$ (12mm$^2$ of L2 cache) and WC2x2 occupies 247mm$^2$ (48mm$^2$ of L2 cache) in 90nm. We speculate that the difference in L2 density is due to additional ports in the EV7 needed to support snooping.

Figure 18 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 21% more AIPC per area as does 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 20% 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 achieves

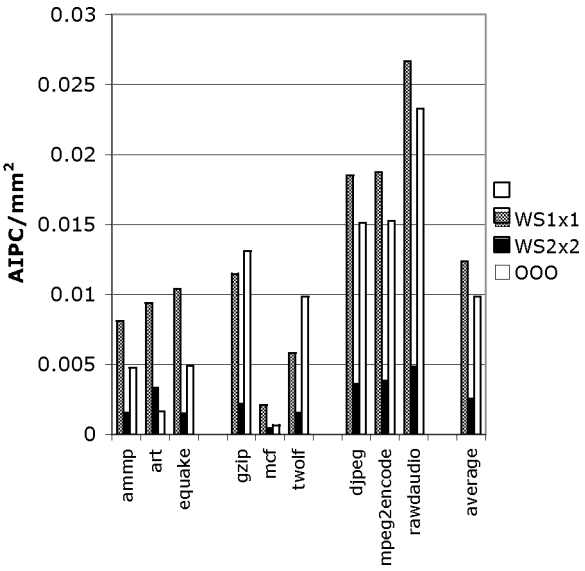


Fig. 18. Performance per unit area. The 1×1 WaveCache is the clear winner in terms of performance per area.

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 solutions 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 lightweight and tightly integrated into the dataflow framework.

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

To evaluate the WaveCache's multithreaded performance, we simulate a 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 that the studies have in common, 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 the figures and examples throughout the rest of this article, 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 them 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 the 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: Respectively, these are 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 19 illustrates how, using the new instructions, thread $t$ creates a new thread $s$, and how 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 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. In the best case, it takes just a few cycles to spawn a thread in the WaveCache, but the average cost depends on several issues, including contention in the network and for store buffer resources. To assess this overhead empirically, 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

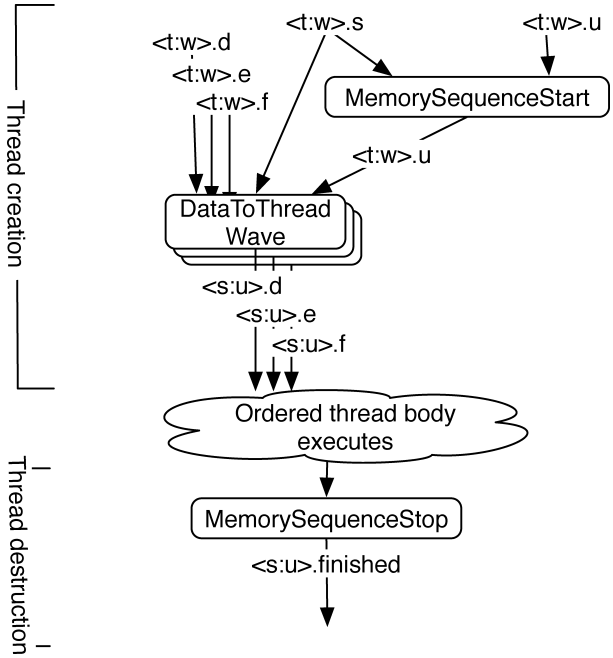


Fig. 19. Thread creation and destruction. Thread $t$ spawns a new thread $s$ by sending $s$'s THREAD-ID $s$ and WAVE-NUMBER $u$ to MEMORY-SEQUENCE-START, which allocates a store buffer to handle the first wave in the new thread. The result of the MEMORY-SEQUENCE-START instruction helps to trigger the 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.

parallel, can the WaveCache speed-up execution by spawning a new thread for every loop iteration?

Figure 20 is a contour plot of speedup of the loop as a function of its loop size (critical path length in ADD instructions is on 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 shows 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 (labeled "B"). Increasing the number of instructions to 20 quadruples performance ("C").

## 5.2 Synchronization

The ability to efficiently create and terminate pthread-style threads [Nichols et al. 1996], as described in the previous subsection, provides only part

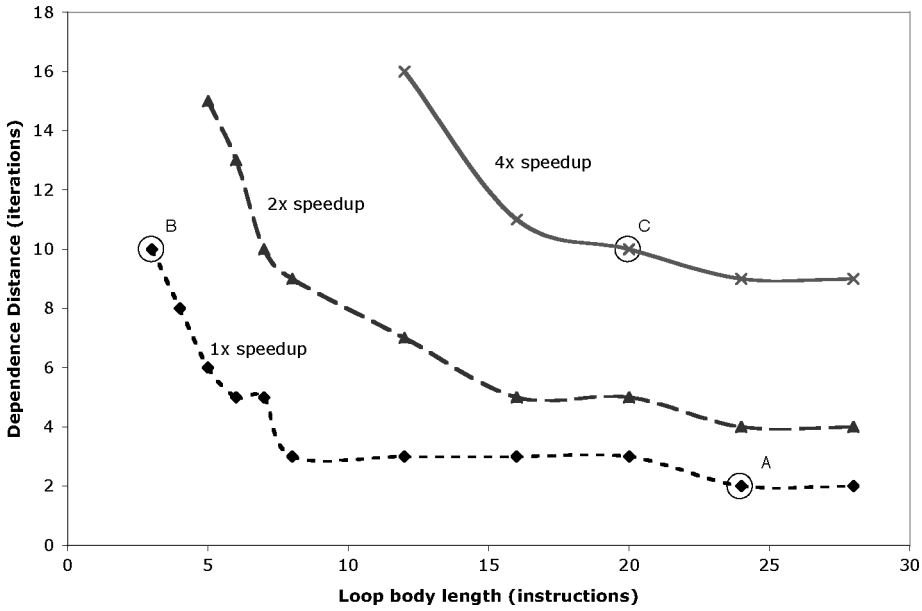


Fig. 20. Thread creation overhead. Contour lines for speedups of 1× (no speedup), 2×, and 4×. The area above each line is a region of program speedup at or above the stated value. Spawning wave-ordered threads in the WaveCache is lightweight enough to profitably parallelize loops with as few as ten instructions in the loop body if four independent iterations execute.

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 ones. In some situations, such as prior to 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, namely 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 [Adve and Gharachorloo 1996]. 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 [Goodman et al. 1989;

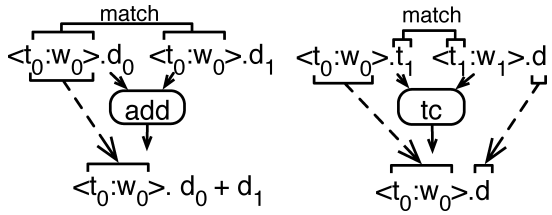


Fig. 21. Tag matching. Most instructions, like the ADD shown here at left, fire when the thread and wave numbers on both input tokens match. Inputs to THREAD-COORDINATE (right) match if the THREAD-ID of the token on the second input matches the data value of the token on the first input.

Tullsen et al. 1999]. 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 [Keckler et al. 1998; Barth et al. 1991], but tailored to WaveScalar's dataflow framework.

As Figure 21 illustrates, THREAD-COORDINATE requires slightly different matching rules.[4] All WaveScalar instructions *except* THREAD-COORDINATE fire when the tags of two input values match, and they produce outputs with the same tag (Figure 21, 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 21, 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 21, this produces an output of $<t_0, w_0>.d$. In essence, THREAD-COORDINATE passes the second input's value $d$ to the thread of the first input $t_0$. Since the two inputs come from different threads, this forces the receiving thread ($t_0$ in this case) to wait for the data from the sending thread $t_1$ before continuing execution.

To support THREAD-COORDINATE in hardware, we augment the tag-matching logic at each PE. We add two counters at each PE to relabel the WAVE-NUMBERs of the inputs to THREAD-COORDINATE instructions so that 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 it is possible construct many kinds of synchronization objects using THREAD-COORDINATE, for brevity we only illustrate a simple mutex (see

[4]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 [Shimada et al. 1984]. Monsoon's M-structure store units had a special matching rule to enforce load-store order [Papadopoulos and Traub 1991].

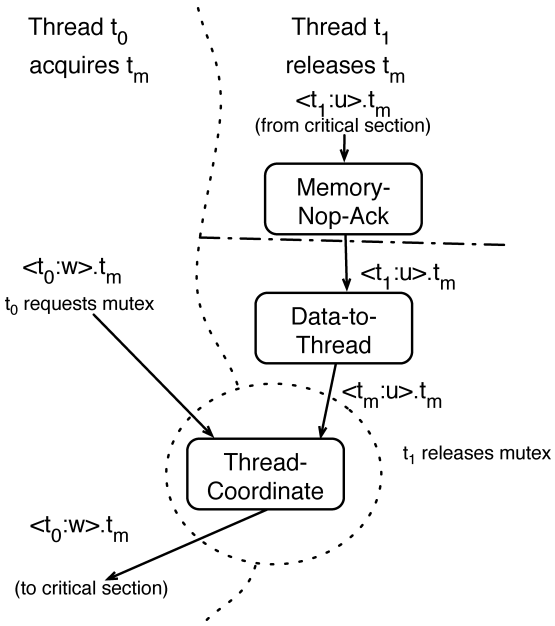


Fig. 22. A mutex. THREAD-COORDINATE is used to construct a mutex, and the value $t_m$ identifies the mutex.

Figure 22), although we have also implemented barriers and conditional variables. 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 22 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 the figure). First, $t_1$ ensures that the memory operations it executed inside the critical section have completed by executing MEMORY-NOP-ACK. Then, $t_1$ uses DATA-TO-THREAD to create the token $<t_m, u>.t_m$, which it sends to the second input port of THREAD-COORDINATE, thereby releasing the mutex.

Another thread ($t_0$ in the figure) can attempt to acquire the mutex by sending $<t_0, w>.t_m$ (the data is the mutex) to THREAD-COORDINATE. This token will either find the token from $t_1$ waiting for it (i.e., the lock is free) or await its arrival (i.e., $t_1$ still holds the lock). When the release token from $t_1$ and the request token from $t_0$ are both present, THREAD-COORDINATE will find that they match according to the rules discussed previously, 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 (see Table III). We use the toolchain and simulator described in Section 4.1.

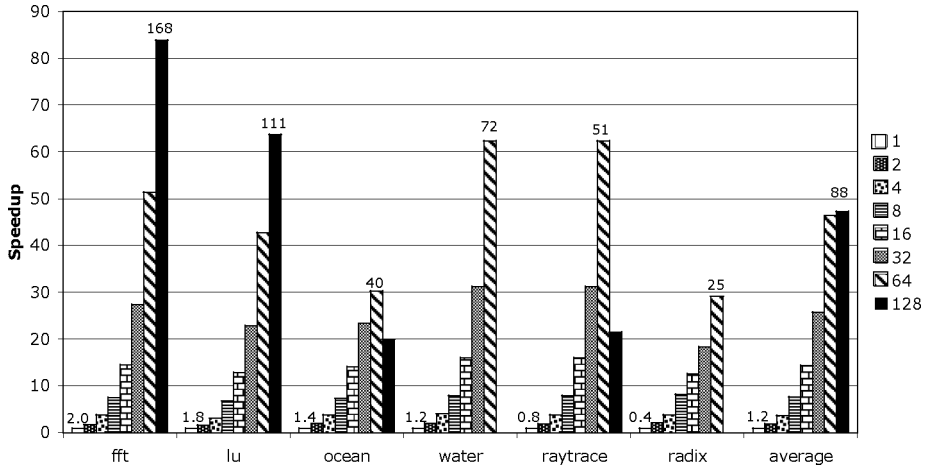


Fig. 23. Splash-2 on the WaveCache. We evaluate each of our Splash-2 benchmarks on the baseline WaveCache with between 1 and 128 threads. The bars represent speedup in total execution time. The numbers above the single-threaded bars are the IPC for that configuration. Two benchmarks, *water* and *radix*, cannot utilize 128 threads with the input dataset we use, so that value is absent.

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$290mm$^2$, with an on-chip 16MB L2.

After skipping past initialization, we measure the execution of 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 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 23 contains speedups of multithreaded WaveCaches for all six benchmarks as compared to their single-threaded running times. On average, the WaveCache achieves near-linear speedup (25$\times$) for up to 32 threads. Average performance increases sub-linearly with 128 threads, up to 47$\times$ speedup with an average IPC of 88.

Interestingly, increasing beyond 64 threads for *ocean* and *raytrace* reduces performance. The drop-off occurs because of WaveCache congestion from the larger instruction working sets and L1 data evictions due to capacity misses. For example, going from 64 to 128 threads, *ocean* suffers 18% more WaveCache instruction misses than would be expected from the additional compulsory misses. In addition, the operand-matching cache miss rate increases by 23%. Finally, the data cache miss rate, 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 11. Since the applications scale almost linearly, this data demonstrates that the reduced performance is due primarily to increased contention for WaveCache resources.

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 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 Lo et al. [1997] and Ekman and Stenström [2003]. Comparing data from such diverse sources is difficult, and drawing precise conclusions about the results is not possible; however, we believe that the measurements are 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 16MB L2 cache distributed in banks around the edge of the chip 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 ballpark with respect to size.

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

Figure 24 shows the results. Stacked bars represent the increase in performance contributed by executing with more threads. The bars labeled *ws* depict the performance of the WaveCache. The bars labeled *scmp* represent the performance of a CMP whose cores are the scalar processors described previously 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 four-state snoopy protocol. Up to four 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 [Hammond et al. 2000] and a single Piranha chip [Barroso et al. 2000], respectively.

The bars labeled *smt8*, *cmp4*, and *cmp2* are the 8-threaded SMT and 4- and 2-core out-of-order CMPs from Lo et al. [1997]. 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 instruction and data caches.

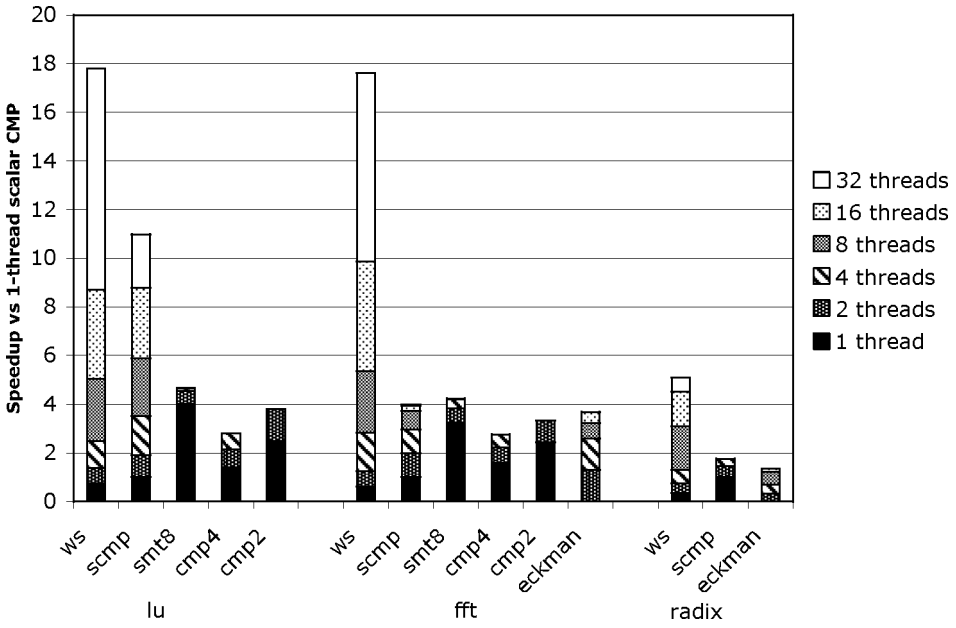


Fig. 24. Performance comparison of various architectures. Each bar represents performance of a given architecture for between 1 and 32 threads. We normalize running times to that of a single-issue scalar processor with a high memory access latency, and compare speedups of various multithreaded architectures. Specifically, *ws* is a 4 × 4 WaveCache, and *scmp* is a CMP of the aforementioned scalar processor on a shared bus with MESI coherence. Respectively, *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 Lo et al. [1997]. Finally, *ekman* [Ekman and Stenström 2003] 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.

To compare the results from Ekman and Stenström [2003] (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 Ekman and Stenström [2003], 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 WaveCache can outperform the other architectures at high thread counts. It executes 1.9× to 6.13× faster than *scmp*, 1.16× to 1.56× faster than *smt8*, and 5.7× to 16× faster than the various out-of-order CMP configurations. Component metrics show that 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 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. It is likely that future CMPs will address this limitation by moving away from bus-based interconnects, and it would be interesting to compare the performance of those systems with the WaveCache. For other von Neumann CMP systems, the fixed allocation of execution resources is the limit [Lo et al. 1997], resulting in a decrease in per-processor IPC. For example, in *ekman*, the per-processor IPC drops by 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 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 multithreaded 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 this is not the limit of what WaveScalar can do.

This section exploits WaveScalar's dataflow underpinning to achieve two things that conventional von Neumann machines cannot. Firstly, 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 wave-ordering hardware. Because 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 next, a

programmer can restrict this ordering by adding edges to a program's dataflow graph.

Secondly, the WaveCache can support very fine-grain threads. On von Neumann machines the amount of hardware devoted to a thread is fixed (e.g., one core on a CMP or one thread context on an SMT machine), and the number of threads that can execute at once is relatively small. On 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 inexpensive. As a result, it is feasible to break a program up into hundreds of parallel fine-grain threads.

We begin by describing the unordered memory interface. Then we use it in addition to 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 and fine-grain dataflow-style threads, we can achieve performance greater than utilizing either alone, in this case, a 9× 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 can only express limited parallelism (i.e., by using ripple numbers). WaveScalar's unordered interface makes a different tradeoff: It cannot efficiently provide the sequential ordering that conventional programs require, but excels at expressing parallelism because it eliminates unnecessary ordering constraints and avoids contention for the store buffer. Accordingly, 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 neither directly nor 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 that 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,

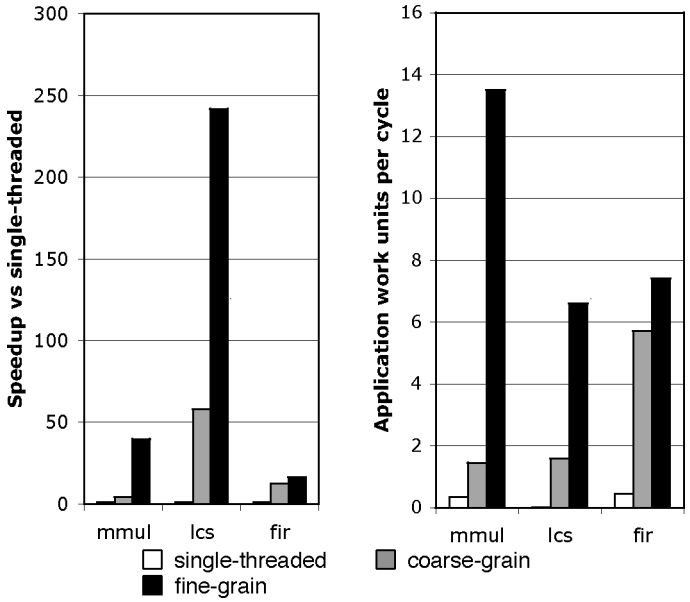


Fig. 25. Fine-grain performance. These graphs compare the performance of our three implementation styles. The graph on the left shows execution-time speedup relative to the serial coarse-grain implementation. The graph on the right compares the work per cycle achieved by each implementation measured in multiply-accumulates for MMUL and FIR and in character comparisons for LCS.

since its only purpose is to signal that the store has completed. 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 array tile size to achieve the best performance possible for a particular implementation. MMUL multiplies $128 \times 128$ entry matrices, LCS compares strings of 1,024 characters, and FIR filters 8,192 inputs with 256 taps. They use between 32 (FIR) and 1,000 (LCS) threads. Each version is run to completion.

Figure 25 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, unordered implementations can use more threads. It would be easy to write a pthread-based version that spawns hundreds or thousands of threads, but the WaveCache cannot execute this 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, fine-grain unordered implementations exploit more inter- and intrathread parallelism, allowing them to exploit many PEs at once. In fact, the numerous threads that each kernel spawns can easily fill and utilize the entire WaveCache. MMUL is the best example; it executes 27 memory operations per cycle on average (about one per every two clusters), compared to just six 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 and then pass them from thread to thread. For these two applications the limiting factor is intercluster network bandwidth. Both algorithms involve a great deal of interthread communication, and since the computation uses the entire $8 \times 8$ array of clusters, intercluster communication is unavoidable. For LCS, 27% of the messages travel across the intercluster 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 further improve performance 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.

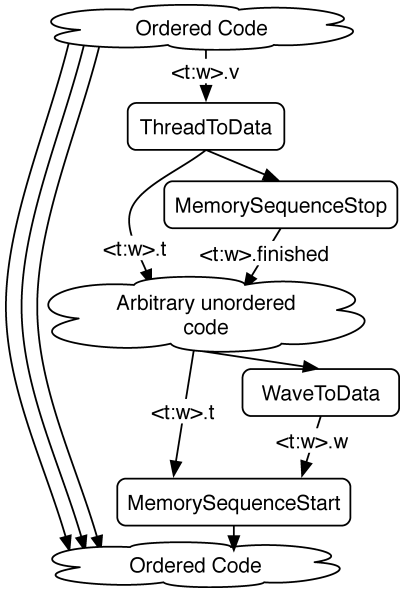


Fig. 26. Transitioning between memory interfaces. The transition from ordered to unordered memory and back again.

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 ordered and unordered interfaces to exist simultaneously.

—*Example* 1. Figure 26 shows a code sequence that transitions from wave-ordered memory to unordered memory and back again. The process is quite similar to terminating and restarting a pthread-style thread. At the end of the ordered code, a THREAD-TO-DATA instruction extracts the current THREAD-ID, and a MEMORY-SEQUENCE-STOP instruction terminates the current memory ordering. MEMORY-SEQUENCE-STOP outputs a value, labeled *finished* in the figure, after all preceding wave-ordered memory operations have completed. The *finished* token triggers the dependent unordered memory operations, ensuring that they do not execute until the preceding 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.

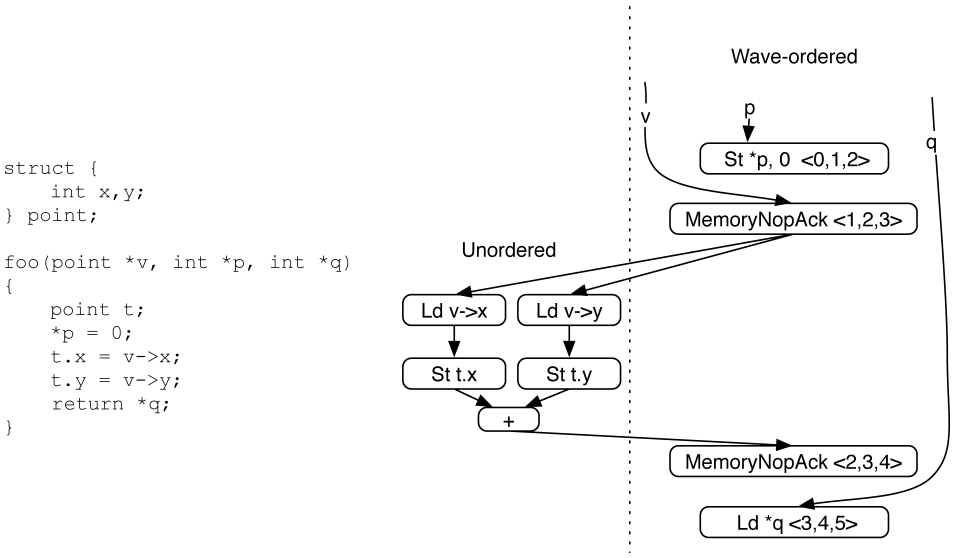


Fig. 27. Using ordered and unordered memory together. A simple example where MEMORY-NOP-ACK is used to combine ordered and unordered memory operations to express memory parallelism.

Figure 27 shows how the MEMORY-NOP-ACK instruction from Section 5.2.1 allows programs to take advantage of this technique. Recall that MEMORY-NOP-ACK is a wave-ordered memory operation that operates like a memory fence instruction, returning a value when it completes. We use it here to synchronize ordered and unordered memory accesses. In function `foo`, the loads and stores that copy `*v` into `t` can execute in parallel, but must wait for the store to `p`, which could point to any address. Likewise, the load from address `q` cannot proceed until the copy is complete. The wave-ordered memory system guarantees that the store to `p`, the two MEMORY-NOP-ACKs, and the load from `q` fire in the order shown (top-to-bottom). The data dependences between the first MEMORY-NOP-ACK and the unordered loads at left ensure that the copy occurs after the first store. The ADD instruction simply coalesces the outputs from the two STORE-UNORDERED-ACK instructions into a trigger for the second MEMORY-NOP-ACK which ensures that 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 benchmark suite. *equake* spends most of its time in the function *smvp*, with the bulk of its remainder confined to a single loop in the program's *main* function. In the discussion to follow, 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 a particular value at-a-time. This idiom is inspired by M-structures [Barth et al. 1991], 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 multithreaded 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, namely 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, a lightweight memoryless synchronization primitive, and a memory fence that provides a relaxed consistency model. For finer-grain 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 design makes it scalable and significantly reduces the architecture's complexity. Our RTL model shows that a WaveCache capable of efficiently running real-world multithreaded applications would occupy only 253mm$^2$ in currently available process technology, while a single-threaded version requires only 48mm$^2$.

Our experimental results show that the WaveCache performs comparably to a modern out-of-order design for single-threaded codes and provides

21% more performance per area. For multithreaded Splash2 benchmarks, the WaveCache achieves 30–83× speedup over single-threaded versions, and outperforms a range of von Neumann-style multithreaded processors. By exploiting our new unordered memory interface, we demonstrated that hundreds of fine-grain threads on the WaveCache can complete 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


ADVE, S. V. AND GHARACHORLOO, K. 1996. Shared memory consistency models: A tutorial. *Comput. 29*, 12, 66–76.

AGARWAL, V., HRISHIKESH, M. S., KECKLER, S. W., AND BURGER, D. 2000. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*. ACM Press, New York. 248–259.

ARVIND. 2005. Dataflow: Passing the token. *ISCA* keynote in *Annual International Symposium on Computer Architecture*.

ARVIND, NIKHIL, R. S., AND PINGALI, K. K. 1989. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst. 11*, 4, 598–632.

BARROSO, L. A., GHARACHORLOO, K., MCNAMARA, R., NOWATZYK, A., QADEER, S., SANO, B., SMITH, S., STETS, R., AND VERGHESE, B. 2000. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*. ACM Press, New York. 282–293.

BARTH, P. S., NIKHIL, R. S., AND ARVIND. 1991. M-structures: Extending a parallel, non-strict, functional languages with state. Tech. Rep. MIT/LCS/TR-327, Massachusetts Institute of Technology. March.

BECK, M., JOHNSON, R., AND PINGALI, K. 1991. From control flow to data flow. *J. Parallel Distrib. Comput. 12*, 2, 118–129.

BUDIU, M., VENKATARAMANI, G., CHELCEA, T., AND GOLDSTEIN, S. C. 2004. Spatial computation. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, New York. 14–26.

CADENCE. 2007. Cadence website. http://www.cadence.com.

CHRYSOS, G. Z. AND EMER, J. S. 1998. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA. 142–153.

CULLER, D. E. 1990. Managing parallelism and resources in scientific dataflow programs. Ph.D. thesis, Massachusetts Institute of Technology.

CULLER, D. E., SAH, A., SCHAUSER, K. E., VON EICKEN, T., AND WAWRZYNEK, J. 1991. Fine-Grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, New York. 164–175.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4, 451–490.

DALLY, W. J. AND SEITZ, C. L. 1987. Deadlock-Free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput. 36*, 5, 547–553.

DAVIS, A. L. 1978. The architecture and system method of DDM1: A recursively structured data driven machine. In *Proceedings of the 5th Annual Symposium on Computer Architecture (ISCA)*. ACM Press, New York. 210–215.

DENNIS, J. B. AND MISUNAS, D. P. 1975. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture (ISCA)*. ACM Press, New York. 126–132.

DESIKAN, R., BURGER, D. C., KECKLER, S. W., AND AUSTIN, T. M. 2001. Sim-Alpha: A validated, execution-driven Alpha 21264 simulator. Tech. Rep. TR-01-23, University of Texas-Austin, Department of Computer Sciences.

EKMAN, M. AND STENSTRÖM, P. 2003. Performance and power impact of issue width in chip-multiprocessor cores. In *Proceedings of the International Conference on Parallel Processing*.

FEO, J. T., MILLER, P. J., AND SKEDZIELEWSKI, S. K. 1995. SISAL90. In *Proceedings of the Conference on High Performance Functional Computing*.

GOLDSTEIN, S. C. AND BUDIU, M. 2001. NanoFabrics: Spatial computing using molecular electronics. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*. ACM Press, New York. 178–191.

GOODMAN, J. R., VERNON, M. K., AND WOEST, P. J. 1989. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, New York. 64–75.

GRAFE, V. G., DAVIDSON, G. S., HOCH, J. E., AND HOLMES, V. P. 1989. The Epsilon dataflow processor. In *Proceedings of the 16th Annual International Symposium on Computer Architecture (ISCA)*. ACM Press, New York. 36–45.

GURD, J. R., KIRKHAM, C. C., AND WATSON, I. 1985. The Manchester prototype dataflow computer. *Commun. ACM 28*, 1, 34–52.

HAMMOND, L., HUBBERT, B. A., SIU, M., PRABHU, M. K., CHEN, M., AND OLUKOTUN, K. 2000. The Stanford Hydra cmp. *IEEE Micro. 20*, 2, 71–84.

HRISHIKESH, M. S., BURGER, D., JOUPPI, N. P., KECKLER, S. W., FARKAS, K. I., AND SHIVAKUMAR, P. 2002. 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 (ISCA)*. IEEE Computer Society, Los Alamitos, CA. 14–24.

JAIN, A. E. A. 2001. A 1.2GHz Alpha microprocessor with 44.8GB/s chip pin bandwidth. In *Proceedings of the IEEE International Solid-State Circuits Conference*. Vol. 1. 240–241.

KECKLER, S. W., DALLY, W. J., MASKIT, D., CARTER, N. P., CHANG, A., AND LEE, W. S. 1998. Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA. 306–317.

KISHI, M., YASUHARA, H., AND KAWAMURA, Y. 1983. Dddp-A distributed data driven processor. In *Proceedings of the 10th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society Press, Los Alamitos, CA. 236–242.

KREWEL, K. 2005. Alpha EV7 processor: A high-performance tradition continues. Microprocessor Rep.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO)*. 330–335.

LEE, W., BARUA, R., FRANK, M., SRIKRISHNA, D., BABB, J., SARKAR, V., AND AMARASINGHE, S. 1998. 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)*. ACM Press, New York. 46–57.

LO, J. L., EMER, J. S., LEVY, H. M., STAMM, R. L., TULLSEN, D. M., AND EGGERS, S. J. 1997. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst. 15*, 3, 322–354.

MAHLKE, S. A., LIN, D. C., CHEN, W. Y., HANK, R. E., AND BRINGMANN, R. A. 1992. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society Press, Los Alamitos, CA. 45–54.

MAI, K., PAASKE, T., JAYASENA, N., HO, R., DALLY, W. J., AND HOROWITZ, M. 2000. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*. ACM Press, New York. 161–171.

MERCALDI, M., SWANSON, S., PETERSEN, A., PUTNAM, A., SCHWERIN, A., OSKIN, M., AND EGGERS, S. J. 2006a. Instruction scheduling for a tiled dataflow architecture. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 141–150.

MERCALDI, M., SWANSON, S., PETERSEN, A., PUTNAM, A., SCHWERIN, A., OSKIN, M., AND EGGERS, S. J. 2006b. Modeling instruction placement on a spatial architecture. In *Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 158–169.

MUKHERJEE, S. S., WEAVER, C., EMER, J., REINHARDT, S. K., AND AUSTIN, T. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA. 29.

NAGARAJAN, R., SANKARALINGAM, K., BURGER, D., AND KECKLER, S. W. 2001. A design space evaluation of grid processor architectures. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA. 40–51.

NICHOLS, B., BUTTLAR, D., AND FARRELL, J. P. 1996. *Pthreads Programming*. O'Reilly, Sebastopol, CA.

NIKHIL, R. 1990. The parallel programming language id and its compilation for parallel machines. In *Proceedings of the Workshop on Massive Paralleism: Hardware, Programming and Applications*. Acamedic Press.

PAPADOPOULOS, G. M. AND CULLER, D. E. 1990. Monsoon: An explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)*. ACM Press, New York. 82–91.

PAPADOPOULOS, G. M. AND TRAUB, K. R. 1991. Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA)*. ACM Press, New York. 342–351.

PETERSEN, A., PUTNAM, A., MERCALDI, M., SCHWERIN, A., EGGERS, S., SWANSON, S., AND OSKIN, M. 2006. Reducing control overhead in dataflow architectures. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 182–191.

SANKARALINGAM, K., NAGARAJAN, R., LIU, H., KIM, C., HUH, J., BURGER, D., KECKLER, S. W., AND MOORE, C. R. 2003. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*. 422–433.

SHIMADA, T., HIRAKI, K., AND NISHIDA, K. 1984. An architecture of a data flow machine and its evaluation. *ACM SIGARCH Comput. Architecure News 14*, 2, 226–234.

SHIMADA, T., HIRAKI, K., NISHIDA, K., AND SEKIGUCHI, S. 1986. 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 (ISCA)*. IEEE Computer Society Press, Los Alamitos, CA. 226–234.

SMITH, A., GIBSON, J., MAHER, B., NETHERCOTE, N., YODER, B., BURGER, D., MCKINLE, K. S., AND BURRILL, J. 2006. Compiling for edge architectures. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, Los Alamitos, CA. 185–195.

SPEC. 2000. SPEC CPU 2000 benchmark specifications. SPEC2000 Benchmark Release.

SWANSON, S. 2006. The WaveScalar architecture. Ph.D. thesis, University of Washington.

SWANSON, S., MICHELSON, K., SCHWERIN, A., AND OSKIN, M. 2003. Wavescalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA. 291.

SWANSON, S., PUTNAM, A., MERCALDI, M., MICHELSON, K., PETERSEN, A., SCHWERIN, A., OSKIN, M., AND EGGERS, S. J. 2006. Area-Performance trade-offs in tiled dataflow architectures. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA. 314–326.

TAYLOR, M. B., LEE, W., MILLER, J., WENTZLAFF, D., BRATT, I., GREENWALD, B., HOFFMANN, H., JOHNSON, P., KIM, J., PSOTA, J., SARAF, A., SHNIDMAN, N., STRUMPEN, V., FRANK, M., AMARASINGHE, S., AND AGARWAL, A. 2004. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, Los Alamitos, CA. 2.

TSMC. 2007. Silicon design chain cooperation enables nanometer chip design. Cadence Whitepaper. http://www.cadence.com/whitepapers/.

TULLSEN, D. M., LO, J. L., EGGERS, S. J., AND LEVY, H. M. 1999. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA. 54.

Received October 2005; revised October 2006; accepted January 2007