Multigranular Thread Support in WaveScalar

Abstract: *WaveScalar is a recently proposed scalable microarchitecture. The original WaveScalar research developed and evaluated an ISA and microarchitecture that efficiently executes a single, coarse-grain thread. In this paper, we expand that design to support multiple, simultaneously executing threads. Four mechanisms make this possible: (1) instructions that enable and disable wave-ordered memory; (2) additional memory access instructions that bypass wave-ordered memory, exposing additional parallelism; (3) lightweight, interthread synchronization that models hardware queue locks; and (4) a simple, intrathread fence instruction that allows applications to handle relaxed memory consistency.*

We combine these primitives to create a multigranular threading system. By using the expanded memory interface and both synchronization mechanisms, we build a coarse-grain, pthread-style library and execute the Splash2 benchmark suite. These programs achieve a 30-80× performance gain over a single thread on a WaveScalar processor, providing running times that are 10-20× faster than a CMP. Then, using the intra-thread fence and the new memory access instructions, we build very fine-grain threads that achieve 7-14 multiply-accumulates per cycle for well-known kernels. Finally, we show that the two approaches can be combined effectively by using them to parallelize equake*. The integration improved* equake*'s performance by 9× over an optimized serial version.*

1 Introduction

Multithreading is an effective way to improve system performance, and designers have long sought to introduce architectural support for threaded applications [1]. Prior work includes hardware support for multiple thread contexts [2, 3], mechanisms for efficient thread synchronization [4, 5], and consistency models that provide threads with a unified view of memory at lower cost [6]. Because of this large body of work and the large amount of silicon resources available, threaded architectures are now mainstream in commodity systems [7, 8, 9].

Interestingly, no single definition of a thread has proven suitable for all applications. For example, web servers and other task-based systems are suited to coarse-grain, pthread-style threads. Conversely, many media, graphics, matrix, and string algorithms contain significant fine-grain data parallelism. In addition, sophisticated compilers [10, 11, 12] are capable of detecting parallelism on several levels – from instructions to loop bodies to function invocations.

Individual architectures, however, tend not to support this heterogeneity. Threaded architectures usually target a specific thread granularity or, in some cases, a small number of granularities [13, 14, 15], making it difficult or inefficient to execute and synchronize threads of a different grain. For example, extremely fine-grain applications cannot execute efficiently on a shared memory multiprocessor due to the high cost of synchronization [13, 16]. In contrast, dataflow machines provide excellent support for extremely fine-grain threads, but must be programmed in specialized languages to correctly execute traditional coarse-grain applications. This requirement stems from dataflow's inability to guarantee that memory operations will execute in a particular order.

In principle, if it could solve the ordering issue, a dataflow architecture could support a wide range of thread granularities by decomposing coarse-grain threads into fine-grain threads. WaveScalar [17], a recently developed dataflow instruction set, provides one solution, a mechanism called *wave-ordered memory*, which enforces memory order. Consequently, it is the first dataflow architecture that efficiently executes programs written in conventional, imperative languages, like C.

Prior WaveScalar work [17] developed an ISA and microarchitecture to execute a single coarse-grain thread of execution. Here, we expand that design to support multiple threads. We do this with a small set of mechanisms:

- Specific instructions that turn wave-ordered memory on and off. Since each thread in WaveScalar has a separate memory ordering, this is tantamount to creating and terminating coarse-grain threads.

- A simple synchronization primitive that builds a hardware queue lock. This instruction provides memoryless, distributed, interthread synchronization by taking advantage of dataflow's inherent message passing, making it a good match for WaveScalar's distributed microarchitecture.

- A new set of memory operations that enable applications to access memory without adhering to a global memory ordering. These instructions bypass wave-ordered memory, enabling independent memory operations to execute in parallel.

- A dataflow version of a memory fence instruction that allows applications to utilize relaxed consistency models. This instruction also serves as an *intra*thread synchronization mechanism for threads whose memory operations bypass wave-ordered memory.

Taken together, these mechanisms enable WaveScalar to define and differentiate threads with a wide range of granularity. The new wave-ordered memory control mechanisms and the memoryless synchronization primitive, combined with an extended dataflow tag, provide coarse-grain, pthread-style threads. The memory fence mechanism ensures that the execution state of these threads becomes consistent with memory, even under a relaxed consistency model. Using this multithreaded support, the Splash-2 benchmarks executing on WaveScalar achieve speedups of 30-83$\times$ over single-threaded execution.

In addition, WaveScalar uses the memory operations that bypass wave-ordered memory and both synchronization primitives to create extremely small threads. These "unordered threads" have very little overhead and may use very few hardware resources. Hence, they are extremely useful for expressing finer-grain loop and data parallelism. We use them to complete 7-13.5 multiply-accumulates (or similar units of work) per cycle for three commonly used kernels.

Finally, we show that conventional, coarse-grain threads and fine-grain, unordered threads can interact seamlessly in the same application. To demonstrate that integrating both styles is possible and profitable, we apply them to *equake* from the Spec2000 benchmark suite. We parallelize the outermost loop of *equake* with coarse-grain, pthread-style threads and implement a key inner-loop with fine-grain threads that utilize unordered memory. The results demonstrate that the multigranular threading approach achieves significantly better performance than either the coarse- or fine-grain approaches alone.

The remainder of the paper unfolds as follows. We briefly discuss related work on architectural support for multigranular threading and thread support for dataflow machines. Then, because the WaveScalar architecture was just recently introduced, in Section 3 we review those elements of the architecture that are most important for understanding the new support for multiple threads. Next, in Section 4, we introduce and evaluate the mechanisms required to support coarse-grain threads. We follow this in Section 5 with a discussion of the mechanisms needed for fine-grain, unordered threads. Finally, in Section 6, we combine the two threading models in a single application. We conclude the paper in Section 7.

2 Background

Adding thread support to an architecture requires that designers solve several problems. First, they must determine what defines a thread in their architecture. Then, they must simultaneously isolate threads from one another and provide mechanisms, such as access to shared state and synchronization primitives, that allow them to communicate.

Popular multithreaded systems such as SMPs, CMPs [2], and SMTs [3] define a thread in terms of its state: a register set, a program counter, and an address space. In multiprocessors, thread separation is easy, because each thread has its own dedicated hardware and threads can only interact through memory. SMTs and other processors that support multiple thread contexts within a single pipeline (e.g., Tera [9]) must exercise more care to ensure that threads do not interfere with one another. In these architectures, threads can communicate through memory, but other mechanisms are also possible [16].

The $*T$ [18], the J-Machine [14], and the M-machine [19] define threads in similar terms but support two thread granularities. They use fine-grain threads to enable frequent communication (J-machine, $*T$) or hide latency (M-machine). Coarse-grain threads handle long-running, complex computations (J-machine, $*T$) or group fine-grain threads for scheduling (M-machine). Threads communicate via shared memory (J-machine, M-machine [13]), message passing (J-machine), and direct accesses of another thread's registers (M-machine).

Raw [20] offers flexibility in thread definition, communication, and granularity by exposing the communication costs between tiles in a CMP-style grid architecture. A thread's state is at least the architectural state of a single tile, but could include several tiles and their network switches. Threads communicate through shared memory or by writing to the register files of adjacent tiles. For tightly synchronized threads, the compiler can statically schedule communication to achieve higher performance.

TRIPS [21] supports multiple threads by reallocating resources that it would otherwise dedicate to speculatively executing a single thread. In essence, it uses multiple threads to hide memory and branch latencies instead of speculating. The parameters that define a thread remain similar to other architectures.

EM-4 [15, 22] defines a thread using a set of registers and a memory frame. Synchronization is performed in a dataflow style, and the programmer is provided with library routines that make synchronization explicit.

The similarity in thread representation among these architectures reflects their underlying architectures – all are essentially small, register-based, PC-driven fetch-decode-execute-style processors. In contrast, WaveScalar is a dataflow architecture, though not the first to grapple with the role of threads. Most notably, the Monsoon [23, 24], P-RISC [25] and TAM [26] architectures have developed, to different extents, a model of dataflow machines as systems of interacting, fine-grained imperative threads.

P-RISC adapts ideas from dataflow to von Neumann multiprocessors. To this end, it extends a RISC-like instruction set with fork and join instructions and the notion of two-phase memory operations. Programs consist of numerous small imperative threads with small execution contexts. Whenever a thread blocks on a long-latency operation, such as a remote load, another thread is removed from a ready queue (called the token queue) and executes. Synchronization between threads is handled with explicit memory instructions.

Programs for the Monsoon Explicit Token Store (ETS) architecture can be organized as collections of short, von Neumann-style threads that interact with each other and with memory using dataflow-style communication [24]. The technique improves code scheduling by taking advantage of data locality. It also leads to an extension to the architecture in which the short, imperative threads employ a small set of high-speed temporary registers that are not part of the threads' stored context. Synchronization between threads is implicit, through the dataflow firing rule and presence bits in memory.

The Threaded Abstract Machine [26] adapts the Monsoon and P-RISC ideas to take advantage of hierarchical memory and scheduling systems. It does this by allowing the compiler more authority in scheduling code and data, adding a new level of scheduling hierarchy (called a *quantum*), and restricting communication between different groups of threads to well-defined communication interfaces. Synchronization between threads is explicit, as in P-RISC.

As a result of technology scaling curves, assumptions about data locality in dataflow systems must change. Where once a proposed dataflow machine might consist of tens or hundreds of processing and storage elements, each occupying a whole chip or even many chips and circuit boards, future architectures will propose placing hundreds or thousands of such elements on a single microchip. As a result of this basic difference, the mechanisms for threading the WaveCache, although grounded in the work of previous dataflow architectures, must address new problems in innovative ways.

3 WaveScalar review

Before discussing the new elements of the WaveScalar ISA that support multithreading in Section 4, we very briefly review the prior WaveScalar design. This summary contains only those portions of the architecture that provide a context for the threading extensions presented in this paper. For a more in-depth description of WaveScalar in general, see [17].

3.1 The WaveScalar instruction set

In most respects, the WaveScalar instruction set provides the same computing capabilities as a RISC instruction set. Differences occur primarily because it is a dataflow architecture, and with a few notable exceptions, it follows the examples of previous dataflow machines (e.g. [27, 28, 29, 30, 31, 24, 22, 32, 33, 26, 34]).

WaveScalar binaries: A WaveScalar binary is a program's dataflow graph. Each node in the graph is a single instruction which computes a value and sends it to the instructions that consume it. Instructions execute after all input operand values have arrived according to a principle known as the dataflow firing rule [27, 28].

Waves and wave numbers: When compiling a program for WaveScalar, a compiler breaks its control flow graph into pieces called *waves*. The key properties of a wave are: (1) its instructions are partially ordered (i.e., it contains no back-edges), and (2) control enters at a single point. Unlike a similar construct, hyperblocks [35], waves may contain true control-flow joins without predication. Doing so facilitates the easy creation of large waves by unrolling loops.

Multiple waves composed of the same static code (for example, iterations of a loop) may execute simultaneously. To distinguish these instances, known as *dynamic waves*, each value in the WaveScalar ISA carries a tag, called a WAVE-NUMBER. Together, a value and its WAVE-NUMBER form a token. The WaveScalar ISA includes special instructions that manipulate WAVE-NUMBERs. Memory-ordering hardware, described below, constrains the number of simultaneously executing waves, and schedules their memory operations in program order.

Memory ordering: Most programming languages provide the programmer with a model of memory that totally orders memory operations. Lacking an efficient mechanism to support this total load-store ordering, most previous dataflow architectures could not effectively execute programs written in imperative languages (e.g., C, C++, or Java) [36, 37]. [1] WaveScalar overcomes this limitation with a technique called *wave-ordered memory*. In wave-ordered memory, the compiler uses the control flow graph and the instruction order within basic blocks to annotate each memory operation with (1) its position in its wave, called a *sequence number*, and (2) its execution order relative to other memory operations in the same wave. As the memory operations execute, these annotations travel to the memory system, allowing it to apply memory operations in the correct order.

To annotate each memory instruction in a wave, the WaveScalar compiler traverses the wave's control flow graph in

[1] [24] states that Monsoon had a hardware mechanism to guarantee correct ordering between a store and subsequent accesses to the same location, but doesn't describe or evaluate it.

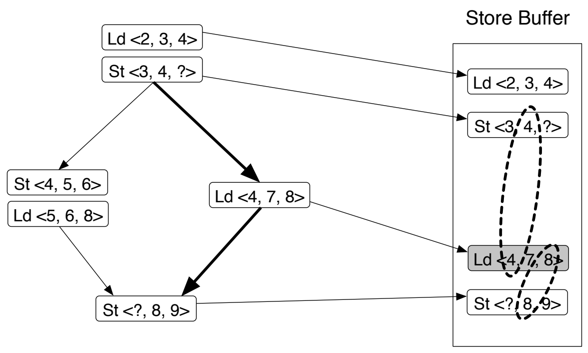


Figure 1: **Annotating memory operations.** A simple wave's control flow graph, showing the memory operations in each basic block, their ordering annotations, and how the annotations allow the store buffer to reconstruct the correct order. The darker arrows show the executed path.

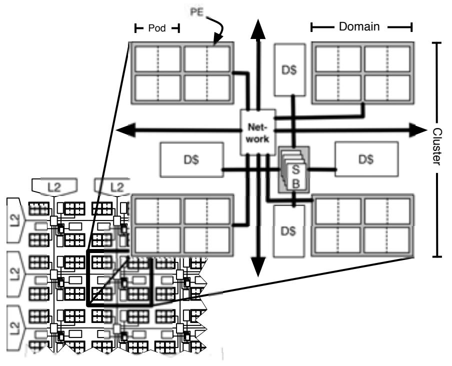


Figure 2: **The WaveCache.** The hierarchical organization of the microarchitecture of the WaveCache.

breadth-first order. Within the basic block at each CFG node, it assigns consecutive sequence numbers to consecutive memory operations. Next, the compiler labels each memory operation with the sequence numbers of its predecessor and successor memory operations, if they can be uniquely determined (see Figure 1, left). Since branch instructions create multiple predecessors or successors, a special wild-card value, '?', is used in these cases. See [17] for details.

During program execution, the memory system (in our implementation, a store buffer) uses these annotations to assemble a wave's loads and stores in the correct order. The right side of Figure 1 shows how the wave-order annotations allow the store buffer to order memory operations and detect those that are missing. Assume the load with sequence number 7 (grayed out) is the last instruction to arrive at the store buffer. Before its arrival, the store buffer knows that at least one memory operation between operations 4 and 8 is missing, because 4's successor and 8's predecessor are both '?'. As a result, it cannot execute operation 8. Operation 7's arrival provides the link between operations 4 and 8, allowing the store buffer to execute operations 7 and 8.

3.2 The WaveCache: a WaveScalar processor

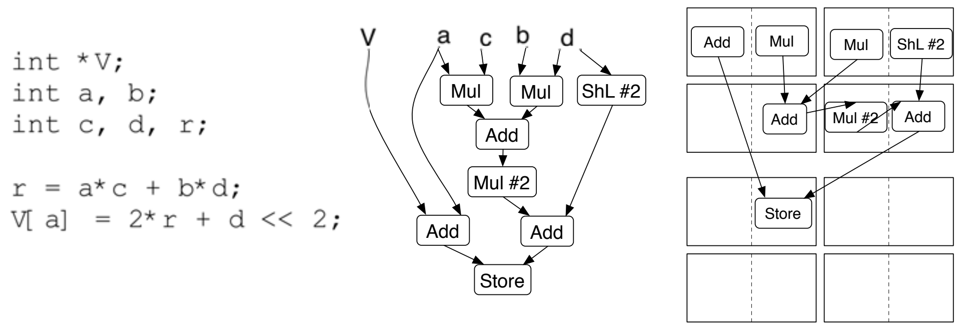
In this section, we summarize the design of a WaveCache that executes WaveScalar binaries and could be built in the near future. This microarchitecture is the baseline model used in the simulations presented in later sections.

Execution: Conceptually, WaveScalar assumes that each static instruction in a program binary executes in a separate processing element (PE). Each PE manages operand tag matching for its instruction. When two operands with identical tags arrive at the PE, the instruction executes (this is the dataflow firing rule [27, 38]) and explicitly communicates the result to statically encoded consumer instructions.

Clearly, building a PE for each static instruction in an application is both impossible and wasteful, so in practice, we dynamically bind instructions to a fixed-size grid of PEs and swap them in and out on demand. We say that the PEs *cache* the working set of the application, hence the name WaveCache. Figure 3 shows a simple code fragment mapped onto part of a WaveCache.

WaveCache Capacity	131, 072 static instructions (64 per PE)		
PEs per Domain	8 (4 pods)	Domains / Cluster	4
PE Input Queue	16 entries, 4 banks	Network Latency	within Pod: 1 cycle
PE Output Queue	8 entries, 4 ports (2r, 2w)		within Domain: 5 cycles
PE Pipeline Depth	5 stages		within Cluster: 9 cycles inter-Cluster: 9 + cluster dist.
L1 Caches	32KB, 4-way set associative, 128B line, 4 accesses per cycle	L2 Cache	16 MB shared, 1024B line, 4-way set associative, 20 cycle access
Main RAM	1000 cycle latency	Network Switch	4-port, bidirectional

Table 1: Microarchitectural parameters of a proposed WaveCache



```
int *V;
int a, b;
int c, d, r;

r = a*c + b*d;
V[ a]  = 2*r + d << 2;
```

Figure 3: **Three views of code in WaveScalar.** At left is the C code for a simple computation. The WaveScalar dataflow graph is shown at center, and the same graph is mapped onto a small patch of the WaveCache substrate at right.

Processor organization: The WaveCache is a grid of simple, 5-stage pipelined processing elements. An RTL model of the design [39] achieves a clock rate of 25 FO4. Each PE contains a functional unit, specialized memories to hold operands, and logic to control instruction execution and communication. A functional unit also contains buffering and storage for several different static instructions, although only one can fire each cycle. Each PE handles tag matching for its own instructions, contributing to the scalability of the WaveCache design.

To reduce communication costs within the grid, PEs are organized hierarchically, as depicted in Figure 2. Two PEs are first coupled, forming a pod that shares operand scheduling and output-to-source bypass logic; within a pod, instructions execute and send their results to the partner PE in a single cycle. Four PE pods comprise a domain, within which producer-consumer latency is five cycles. Four domains are then grouped into a cluster, which also contains wave-ordered memory hardware and a traditional L1 data cache. A single cluster, combined with an L2 cache and traditional main memory is sufficient to run any WaveScalar program. To build larger machines, multiple clusters are connected by an on-chip network and cache coherence is maintained by a traditional, directory-based protocol with multiple readers and a single writer. The coherence directory and the L2 cache are distributed around the edge of the grid of clusters. Table 1 describes the WaveCache configuration used in this paper. Our simulations accurately model contention on all network links and communication busses for operand, memory, and cache coherence traffic. Instruction placement is done on-demand and dynamically snakes instructions across the grid. [2]

Wave-ordered memory hardware: The wave-ordered memory hardware is distributed throughout the WaveCache as part of the store buffers. Each cluster contains four store buffers, all accessed through a single port. A dynamic wave is bound to one store buffer, which fields all memory requests for that wave. The store buffer itself is a small, non-associative memory that holds memory requests. A simple state machine implements the wave-ordered memory logic by “walking” the sequence of requests and stalling when it detects a missing operation. This ensures that memory operations are issued to the L1 data caches in the correct order.

After a wave executes, its store buffer signals the store buffer for the next wave to proceed – analogous to a baton pass in a relay race. This scheme allows all store buffers to remain logically centralized, despite their physically distributed implementation.

The remaining issue lies in assigning store buffers to waves. To accomplish this, we use a table kept in main memory that maps wave numbers to store buffers. Memory instructions send their requests to the nearest store buffer, which accesses the map to determine where the message should go. If the map already has an entry for the current wave, it forwards the message to the appropriate store buffer. If there is no entry, the store buffer atomically updates it with its own location and processes the request.

4 Coarse-grain threads in WaveScalar

In the previous section, we reviewed the WaveScalar instruction set and WaveCache microarchitecture. This prior work [17] described an architecture capable of executing a single coarse-grain thread of execution. In this section, we add support to WaveScalar to simultaneously execute multiple coarse-grain, pthread-style threads. We describe three additions to the ISA and microarchitecture that make this possible. First, we extend the wave-ordered memory interface to support simultaneously active, independent threads of execution. Second, we introduce a lightweight, intrathread synchronization mechanism that enables WaveScalar to provide an efficient relaxed consistency model of memory. Finally, we introduce a low overhead, memoryless synchronization mechanism that models a hardware queue lock and provides efficient intrathread communication.

4.1 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 in a WaveScalar machine with a THREAD-ID in addition to its WAVE-NUMBER. Then, we introduce instructions to associate memory-ordering resources with particular THREAD-IDs. Finally, we make the necessary changes to the WaveCache architecture and

[2]Sigma-1 statically placed instructions[30].

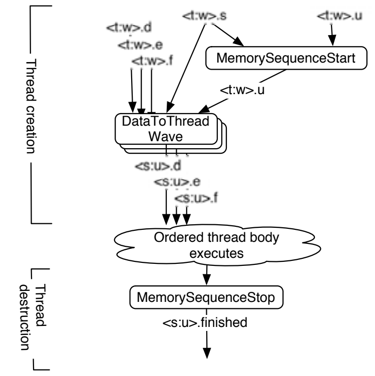


Figure 4: **Thread creation and destruction.** Thread t spawns a new thread s by sending s's THREAD-ID (s) and WAVE-NUMBER (u) to MEMORY-SEQUENCE-START and setting up s's three input parameters with three DATA-TO-THREAD-WAVE instructions. The inputs to each DATA-TO-THREAD-WAVE instruction are a parameter 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 will execute until MEMORY-SEQUENCE-START has finished allocating store buffer area for s. Thread s terminates with MEMORY-SEQUENCE-STOP, whose output token $<s, u>.finished$ indicates that its store buffer area has been deallocated.

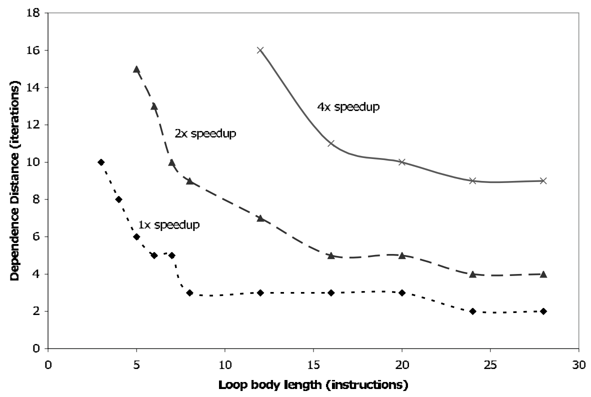


Figure 5: **Thread creation overhead.** Contour lines for speedups of $1\times$ (no speedup), $2\times$ and $4\times$. The area above the each line is a region of program speedup at or above the stated value. Spawning wave-ordered threads in the WaveCache is lightweight enough to profitably parallelize loops with as few as ten instructions in the loop body if four independent iterations may execute.

evaluate their efficiency.

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

To manipulate THREAD-IDS and WAVE-NUMBERS, we introduce several instructions that convert WAVE-NUMBERS and THREAD-IDS to normal data values and back again. The most powerful of these is DATA-TO-THREAD-WAVE which sets both the THREAD-ID and WAVE-NUMBER at once; DATA-TO-THREAD-WAVE takes three inputs, $<t_0, w_0>.t_1$, $<t_0, w_0>.w_1$, and $<t_0, w_0>.d$ and produces as output $<t_1, w_1>.d$. WaveScalar also provides two instructions (DATA-TO-THREAD and DATA-TO-WAVE) to set THREAD-IDS and WAVE-NUMBERS separately, as well as two instructions (THREAD-TO-DATA and WAVE-TO-DATA) to extract THREAD-IDS and WAVE-NUMBERS.

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

MEMORY-SEQUENCE-START creates a new wave-ordered memory sequence, i.e., a new thread. This thread is assigned to a store buffer, which services all memory requests tagged with its THREAD-ID and WAVE-NUMBER; requests with the same THREAD-ID but a different WAVE-NUMBER cause a new store buffer to be allocated, as described in Section 3.2.

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 the store buffer resources are released. Figure 4 illustrates how thread t creates a new thread s, thread s executes and then terminates.

Implementation: Adding support for multiple memory orderings requires only small changes to the WaveCache's microarchitecture. First, the widths of the communication busses and operand queues must be expanded to hold THREAD-IDs. Second, instead of storing every 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 instruction to different PEs.

Efficiency: The overhead associated with spawning a thread directly affects the granularity of extractable parallelism. To assess this overhead in the WaveCache, we designed a controlled experiment consisting of a simple parallel loop, in which each iteration executes in a newly spawned thread. We varied the size of the loop body, which effects the granularity of parallelism, and the dependence distance between memory operands, which effects the number of threads which 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 of T iterations (i.e., the ability to execute T iterations in parallel), can I speed up execution by spawning a new thread for every loop iteration?

Figure 5 is a contour plot of speedup of the loop as a function of its loop size (critical path length in ADD instructions, the horizontal axis) and dependence distance (independent iterations, the vertical axis). Contour lines are shown for speedups of $1\times$ (no speedup), $2\times$ and $4\times$. The area above each line is a region of program speedup at or above the labeled value. The data show that the overhead of creating and destroying threads via MEMORY-SEQUENCE-START and MEMORY-SEQUENCE-STOP 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 dependence distance of 10 reduces the size of profitably parallelizable loops to only 4 instructions. Increasing the number of instructions to 20 quadruples performance. (If independent iterations need to make potentially recursive function calls, extra overhead may apply.)

4.2 Synchronization

The ability to efficiently create and terminate pthread-style threads, as described in the previous subsection, provides only part of the functionality required to make multithreading useful. Independent threads must also synchronize and communicate with one another. WaveScalar recognizes two types of synchronization: intra- and interthread. Intrathread synchronization is comparable to a memory fence or barrier. It is used to build a relaxed consistency model by synchronizing the execution of a thread with its outstanding memory operations. The second primitive models a hardware queue lock and provides interthread synchronization. In the following subsections, we discuss the mechanisms that support these two forms of synchronization and then follow with an example mutex.

4.2.1 Intrathread synchronization

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

Multiprocessors provide a variety of relaxed consistency models, such as those described in [40]. Some, including release consistency [6] and the model used by the Alpha [41], ensure a consistent view only in the presence of memory barrier instructions. MEMORY-NOP-ACK provides this functionality by forcing a thread's memory operations to memory.

4.2.2 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 [42, 16]. 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 [13, 43], but is tailored to WaveScalar's dataflow framework. Rather than utilize an additional hardware memory and finite state machine to implement it, we exploit the tag matching logic used by every processing element to carry out dataflow execution.

Figure 6 illustrates the matching rules required to support THREAD-COORDINATE and how they differ from the matching rules for normal instructions.[3] All WaveScalar instructions *except* THREAD-COORDINATE fire when the tags of two input values match, and they produce outputs with the same tag (Figure 6, left). For example, in the figure, both the input tokens

[3]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[44]. Monsoon's M-structure store units had a special matching rule to enforce load-store order[24].

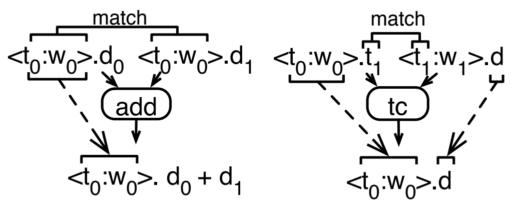


Figure 6: **Tag matching.** Most instructions, like the ADD shown here at left, fire when the thread and wave numbers on both input tokens match. Inputs to THREAD-COORDINATE (right) match if the THREAD-ID of the token on the second input matches the data value of the token on the first input.

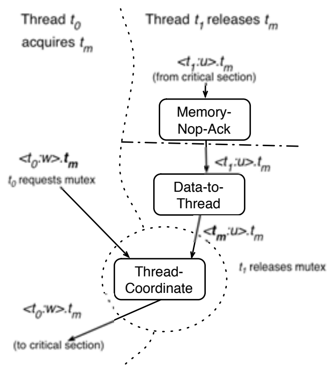


Figure 7: **A mutex.** THREAD-COORDINATE is used to construct a mutex, as described in the text.

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 6, where the data value of the left input token and the thread value 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 6, this produces an output of $<t_0, w_o>.d$. 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 a message from the sending thread (t_1) before continuing execution.

Though one can implement many kinds of synchronization objects using THREAD-COORDINATE, for brevity we only illustrate how THREAD-COORDINATE is used to construct a simple mutex (Figure 7). 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 7 is represented by a THREAD-ID, t_m, although it is not a thread in the usual sense; instead, t_m's sole function is to uniquely name the mutex. A thread t_1 that has locked mutex t_m releases it in two steps (right side of figure). First, t_1 ensures that the memory operations it executed inside the critical section have completed by executing MEMORY-NOP-ACK. Then, t_1 uses DATA-TO-THREAD to create the token $<t_m, u>.t_m$, which it sends to the second input port of THREAD-COORDINATE, thereby releasing the mutex.

This token waits on THREAD-COORDINATE's second input port until another thread, t_0 in the figure, attempts to acquire the mutex. When this happens, t_0 sends the a token $<t_0, w>.t_m$ (whose data is the mutex) to THREAD-COORDINATE. By the rules discussed above, this token matches that sent by t_1, causing THREAD-COORDINATE to 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.

4.3 Splash-2

In this section, we evaluate WaveScalar's new multithreading facilities by executing coarse-grain, multithreaded applications from the Splash-2 benchmark suite. We compiled each application with the DEC cc compiler using -O4 optimizations. A

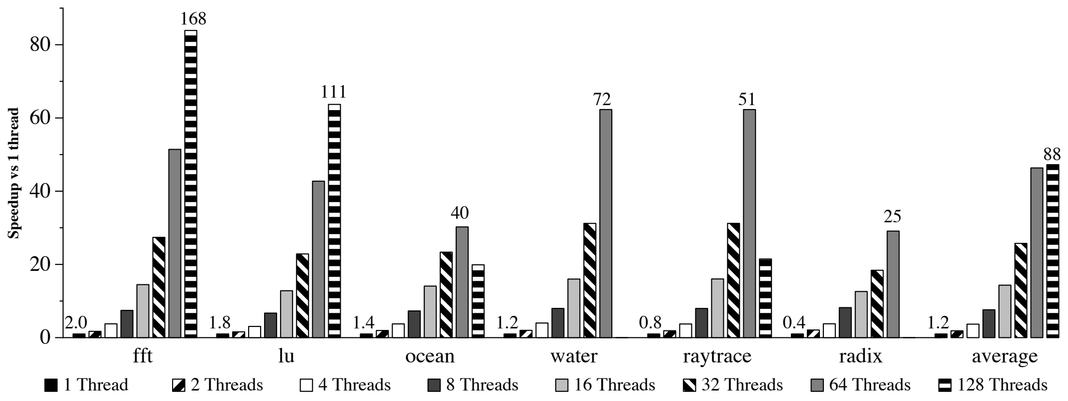


Figure 8: **Splash-2 on the WaveCache.** We evaluate each of our Splash-2 benchmarks on the baseline WaveCache with between 1 and 128 threads. The bars represent speedup in total execution time. The numbers above the single-threaded bars are IPC for that configuration. Two benchmarks, *water* and *radix*, cannot utilize 128 threads with the input data set we use, so that value is absent.

binary translator-based toolchain was used to convert these binaries into WaveScalar executables and to integrate our memoryless synchronization library. We executed the applications on an execution-driven simulator, which models the WaveCache architecture described in Section 3.2. We skip past initialization and execute the parallel phases of the benchmarks.

Our performance metric is execution-time speedup relative to a single thread executing on the WaveCache. We also compare the WaveScalar speedups to those calculated by other researchers for other threaded architectures. Component metrics help explain these bottomline results, where appropriate.

Evaluation of a multithreaded WaveCache. Figure 8 contains speedups of multithreaded WaveCaches for all six benchmarks, as compared to their single-threaded running time. On average, the WaveCache achieves near-linear speedup (27×) for up to 32 threads. Average performance still increases with 128 threads, but sublinearly, up to 47× speedup with an average IPC of 88.

Interestingly, increasing beyond 64 threads for *ocean* and *raytrace* reduces performance. This is because of WaveCache congestion from their 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 matching cache (used to match operand values for execution) miss rate increases by 23%. Finally, the data cache miss rate, which is essentially constant for up to 32 threads, doubles as the number of threads scales to 128. This additional pressure on the memory system increases *ocean*'s memory access latency by a factor of eleven.

The same factors that cause the performance of *ocean* and *raytrace* to suffer when the number of threads exceeds 64 also reduce the rate of speedup improvement for other applications as the number of threads increases. For example, the WaveCache instruction miss rate quadruples for *lu* when the number of threads dedicated to the computation 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.

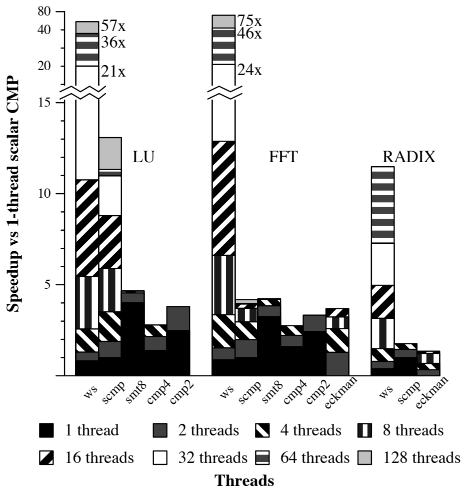


Figure 9: **Performance comparison of various architectures.** Each bar represents performance of a given architecture for a varied number of threads. We normalize running times to that of a single-issue scalar processor with a high memory access latency, and compare speedups of various multithreaded architectures. *ws* is our baseline WaveCache. *scmp* is a CMP of the aforementioned scalar processor on a shared bus with MESI coherence. *smt8*, *cmp4* and *cmp2* are an 8-threaded SMT, a 4-core out-of-order CMP and a 2-core OOO CMP with similar resources, from [45]. *ekman* [46] 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.

Comparison to other threaded architectures. We compared the performance of the WaveCache and a few other architectures on three Splash-2 kernels: LU, FFT and RADIX. (For precise details of these architectures, please see the original papers [45, 46].) To facilitate the comparison of performance numbers from multiple sources, we normalized all performance numbers to the performance of a simulated scalar processor with a 5-stage pipeline. The processor has 16KB data and instruction caches, and a 1MB L2 cache, all 4-way set associative. The L2 hit latency is 12 cycles, and the memory access latency of 1000 cycles matches that of the WaveCache.

Figure 9 shows the results. The stacked bars represent the increase in performance contributed by executing with more threads. The bars labeled *ws* depict the performance of the WaveCache. The bars labeled *scmp* represent the performance of a CMP whose cores are the scalar processors described above. These processors are connected via a shared bus between private L1 caches and a shared L2 cache. Memory is sequentially consistent, and coherence is maintained by a 4-state snoopy protocol. Up to 4 accesses to the shared memory may overlap. For the CMPs the stacked bars represent increased performance from simulating more processor cores. The 4- and 8-stack bars loosely model *Hydra* [47] and a single *Piranha* chip [48], respectively.

The bars labeled *smt8*, *cmp4* and *cmp2* are the 8-threaded SMT and 4- and 2-core out-of-order CMPs from [45]. We extracted their running times from data provided by the authors. Memory latency is low on these systems (dozens of cycles) compared to expected future latencies, and all configurations share the L1 data- and instruction caches.

To compare the results from [46] (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 [46], the authors fixed the execution resources for all configurations, and partitioned them among an increasing number of decreasingly wide CMP cores. For example, the 2-thread component of the *ekman* bars is the performance of a 2-core CMP in which each core has a fetch width of 8, while the 16-thread component represents the performance of 16 cores with a fetch-width of 1. Latency to main memory is 384 cycles, and latency to the L2 cache is 12 cycles.

The graph shows that the WaveCache can handily outperform the other architectures at high thread counts. It executes $4.4\times$ to $18\times$ faster than *scmp*, $5.8\times$ to $18\times$ faster than *smt8*, and $10\times$ to $20\times$ faster than the various out-of-order CMP configurations. Component metrics show that the WaveCache's performance benefits arise from its use of point-to-point communication, rather than a system-wide broadcast mechanism, and from the latency-tolerance of its dataflow execution model. The former enables scaling to large numbers of clusters and threads, while the latter helps mask the increased memory latency incurred by the directory protocol and the high load-use penalty on the L1 data cache.

The performance of all systems eventually plateaus when some bottleneck resource saturates. For *scmp* this resource is shared L1 bus bandwidth. Bus saturation occurs at 16 processors for LU, 8 for FFT and 2 for RADIX [4]. For the other von-Neumann CMP systems, the fixed allocation of execution resources is the limit [45], resulting in a decrease in per-processor IPC. For example, in *ekman*, per-processor IPC drops 50% as the number of processors increases from 4 to 16 for RADIX and FFT. On the WaveCache, speedup plateaus when the working set of all the threads equals the 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.

Visual view of WaveCache execution. Figure 10 shows how multiple threads coexist in the WaveCache. It provides a snapshot of FFT from Splash-2 running with 8 threads. The shade of each processing element denotes the thread that is executing there. The figure illustrates how the execution of each thread and, consequently, its message traffic, are localized in a small region of the WaveCache. It also shows the dynamic allocation of processing resources across threads: each thread occupies a different sized portion of the WaveCache, depending on the number of static instructions it is actively using.

4.4 Discussion

In this section, we extended the WaveScalar architecture to support multiple pthread-style threads by providing support for creating and destroying memory orderings and memoryless synchronization. The result is an efficient threading system that allows multiple coarse-grain threads to execute on a dataflow machine. The mechanisms are lightweight enough that programmers can also use them to express very fine-grain, loop-level parallelism.

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

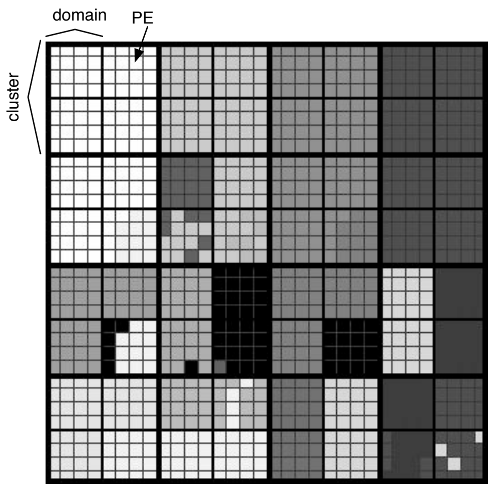


Figure 10: **Splash-2 FFT executing on a small WaveCache.** Each small square is a processing element. Each of the medium and large squares are clusters and domains, respectively. The eight different shades of gray represent eight different threads executing FFT.

Given the mechanisms we have described, it is natural to think of a wave-ordered memory sequence as the essence of a thread, because in most systems the notion of a thread and its memory ordering are inseparable. But in WaveScalar, this perspective is misleading: nothing in the WaveScalar architecture requires a thread to have a memory ordering. If a thread could access memory without interacting with the wave-ordered memory system, it could avoid the serialization bottleneck that a global memory ordering requires. In the next section, we describe an interface to memory that avoids the wave-ordered memory system and show that, combined with fine-grain multithreading, it can provide substantial benefits for applications where a global ordering of memory operations is not necessary for correct execution.

5 Fine-grain, unordered threads

In the previous section, we presented extensions to the WaveScalar instruction set that enable the WaveCache to execute multiple coarse-grain, pthread-style threads simultaneously. The keys to this were extending WaveScalar's tags with THREAD-IDs, providing lightweight memoryless synchronization primitives, and adding management instructions to start and stop ordered memory sequences. The ability to *stop* a memory ordering sequence begs the question, "What if a thread does not have an ordered memory interface at all?" Without an ordered memory interface, WaveScalar threads can execute their memory operations in any order, potentially exposing massive amounts of parallelism. We call these *fine-grain, unordered threads*.

This section develops the notion of fine-grain, unordered threads, describes how they can coexist with the coarse-grain threads of Section 4, and uses them to implement and evaluate three simple kernels. The fine-grain, unordered implementations are up to 9× faster than coarse-grain threaded versions.

5.1 The unordered interface

As described, WaveScalar's original instruction set allows a thread to execute without a memory ordering only if the thread does not access memory. These threads would be more useful if they could safely read and write the same memory used by threads that utilize wave-ordered memory. Then, the coarse-grain threads from the previous section and the new fine-grain, unordered threads could share data through memory.

We provide WaveScalar with a new, unordered interface to memory. This interface does not require a thread to give up all control over the order in which memory instructions execute. Instead, it allows the thread to directly control which memory operations can fire in any order and which must be sequentialized.

To illustrate how WaveScalar accomplishes this, 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. In threads that use wave-ordered memory, the store buffer enforces this constraint; however, since they bypass wave-ordered memory, unordered threads must have a different mechanism.

Dataflow instruction sets like WaveScalar ensure that one instruction executes after another by establishing a data dependence between them. (In our example, this means the load instruction must be data-dependent on the store.) For this technique to work, 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. However, we modify stores to produce a value when they complete.

In addition, the unordered instructions do not carry wave-ordering annotations and bypass the store buffers, accessing the L1 data caches directly. To differentiate the unordered memory operations from their wave-ordered counter-parts, we introduce two unordered operations STORE-UNORDERED-ACK and LOAD-UNORDERED.

5.2 Performance evaluation

To demonstrate the potential of unordered memory in this context, 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 4. *Unordered* uses a single coarse-grain thread written in C to control a pool of fine-grain, unordered threads, written in WaveScalar assembly.

For each application, we tuned the number of threads and the array tile size to achieve the best performance possible for a particular implementation. MMUL multiplies 128×128 entry matrices, LCS compares strings of 1024 characters, and FIR filters 8192 inputs with 256 taps. Each version is run to completion[5].

Figure 11 depicts the performance of each algorithm executing on the WaveCache. 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

[5]The largest problem sizes our simulator could handle (because of memory constraints) did not experience the token explosion seen on previous dataflow machines[31]

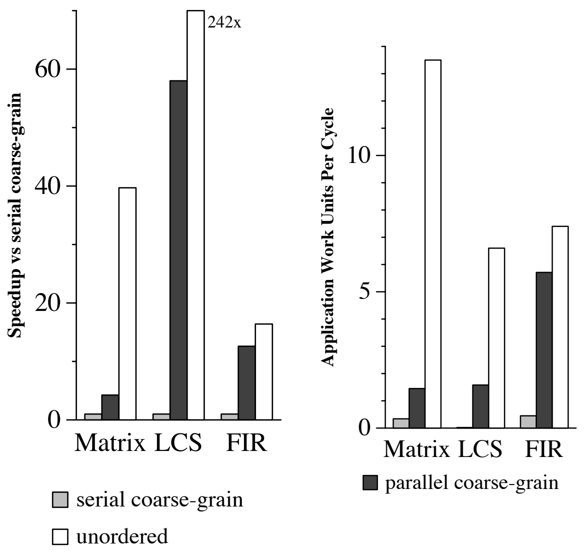


Figure 11: **Fine-grain performance.** These graphs compare the performance of our three implementation styles. The graph on the left shows execution-time speedup relative to the serial coarse-grain implementation. The graph on the right compares the work per cycle achieved by each implementation measured in multiply-accumulates for MMUL and FIR and in character comparisons for LCS.

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 by exploiting more parallelism. Using unordered memory eliminates false dependences, enabling more memory operations to execute in parallel. In addition, bypassing the wave-ordering mechanisms reduces contention for limited store buffer resources. The consequence is a 32-1000$\times$ increase in the number of simultaneously executing threads.

As a result, the fine-grain implementation of MMUL completes 27 memory operations per cycle as compared to 17 per cycle for the coarse-grain implementation.

6 Multigranular threading

In Section 4, 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, unordered threads. 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.

6.1 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.

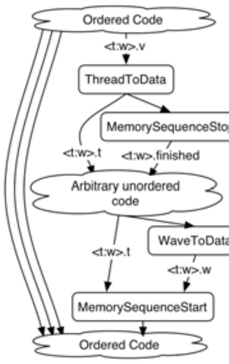
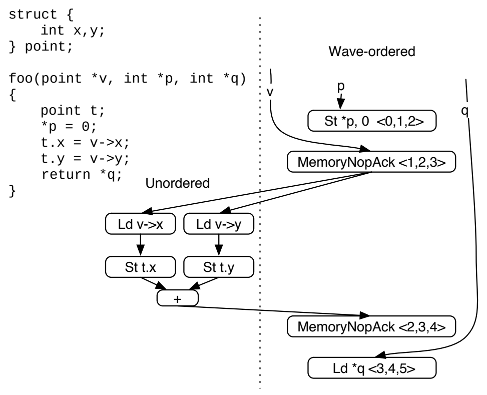


Figure 12: **Transitioning between memory interfaces.** The transition from ordered to unordered memory and back again.



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

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

Figure 13: **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.

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

Example 1: Figure 12 shows a code sequence that transitions from wave-ordered memory to unordered memory and back again. The process is quite similar to terminating and restarting a pthread-style thread. At the end of the ordered code, a THREAD-TO-DATA instruction extracts the current THREAD-ID, and a MEMORY-SEQUENCE-STOP instruction terminates the current memory ordering. MEMORY-SEQUENCE-STOP outputs a value, labeled *finished* in the figure, after all preceding wave-ordered memory operations have completed. The *finished* token triggers the dependent, unordered memory operations, ensuring that they do not execute until the earlier, ordered-memory accesses have completed.

After the unordered portion has executed, a MEMORY-SEQUENCE-START creates a new, ordered memory sequence using the THREAD-ID extracted previously. In principle, the new thread need not have the same THREAD-ID as the original ordered thread. In practice, however, this is convenient, as it allows values to flow directly from the first ordered section to the second (the curved arcs on the left side of the 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 to analyzable operations will simply bypass the wave-ordering interface. This approach allows the two memory interfaces to coexist in the same thread.

Figure 13 shows how the MEMORY-NOP-ACK instruction from Section 4.2 allows programs to take advantage of this technique. In function `foo`, the loads and stores that copy `*v` into `t` can execute in parallel but must wait for the store to `p`, which could point to any address. Likewise, the load from address `q` cannot proceed until the copy is complete. The wave-ordered memory system guarantees that the store to `p`, the two MEMORY-NOP-ACKS, and the load from `q` fire in the order shown (top to bottom). The data dependences between the first MEMORY-NOP-ACK and the unordered loads at left ensure the copy occurs after the first store. The add instruction simply coalesces the outputs from the two STORE-UNORDERED-ACK instructions into a trigger for the second MEMORY-NOP-ACK that ensures the copy is complete before the final load.

6.2 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 [49] benchmark suite. *equake* spends most of its time in the function *smvp*, with the bulk of the remainder confined to a single loop in the program's *main* function. In the discussion below, we refer to this loop in *main* as *sim*.

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

Next, we used the unordered memory interface to exploit fine-grain parallelism in *smvp*. Two opportunities present themselves. First, each iteration of *smvp*'s nested loops loads data from several arrays. Since these arrays are read-only, we used unordered loads to bypass wave-ordered memory, allowing loads from several iterations to execute in parallel. Second, we targeted a set of irregular cross-iteration dependences in *smvp*'s inner loop that are caused by updating an array of sums. These cross-iteration dependences make it difficult to profitably coarse-grain-parallelize the loop. However, the THREAD-COORDINATE instruction lets us extract fine-grain parallelism despite these dependences, since it 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 [43], a dataflow-style memory element. Rewriting *smvp* with unordered memory and M-structures 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 our coarse-grain, pthread-style threads can be used with fine-grain, unordered threads to accelerate a single application.

7 Conclusion

This paper has presented and evaluated a multithreaded design that can profitably execute threads of sizes ranging from as large as a process to as small as a handful of instructions. Such a wide range of thread granularity is possible or, more to the point, such small threads are possible, because WaveScalar drastically reduces the overhead normally required to manipulate threads. Although different aspects of the design contribute to the low overhead, the underlying reason that these different thread functions are so cheap is that they operate within the context of WaveScalar's dataflow architecture that is built atop a distributed dataflow microarchitecture, the WaveCache.

In this paper, we described four extensions to the previous WaveScalar ISA: (1) an extended tag space and instructions that place the wave-ordered memory interface under application control, (2) a lightweight, memoryless synchronization primitive used for interthread communication, (3) a simple dataflow-compatible memory fence-like instruction that synchronizes the execution of a thread with the state of the wave-ordered interface, and (4) a new set of memory instructions that bypass the wave-ordering mechanism and enable threads to control the order of memory access directly.

We combined the extended wave-ordering interface and synchronization mechanisms to provide support for simultaneous execution of multiple coarse-grain, pthread-style threads. Experimental results demonstrate a 30-83× speedup on codes from the Splash2 benchmark suite. By exploiting our new unordered memory interface, we demonstrated how hundreds of fine-grain threads on the WaveCache can complete 13 multiply-accumulates per cycle for selected algorithm kernels. Finally, we combined all of our new mechanisms and threading models to create a multigranular parallel version of equake which is faster than either threading model alone.

References

[1] J. E. Thornton, "Parallel operation in the control data 6600," *Fall Joint Computers' Conference*, vol. 26, pp. 33–40, 1961.

[2] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," in *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[3] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 25th International Symposium on Computer Architecture*, 1995.

[4] A. Kagi, D. Burger, and J. R. Goodman, "Efficient synchronization: Let them eat QOLB," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 170–180, 1997.

[5] J. R. Goodman, M. K. Vernon, and P. J. Woest, "Efficent synchronization primitives for large-scale cache-coherent multiprocessors," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 64–75, 1989.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15–26, 1990.

[7] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the pentium 4 processor," *Intel Technology Journal*, 2001.

[8] M. Funk, "Simultaneous multi-threading (SMT) on eServer iSeries POWER5 processors," May 2004. IBM Corporation.

[9] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera computer system," in *Proceedings of the1990 International Conference on Supercomputing*, pp. 1–6, 1990.

[10] M. Hall *et al.*, "Maximizing multiprocessor performance with the SUIF compiler," *Computer*, December 1996.

[11] K. McKinley, *Automatic and Interactive Parallelization*. PhD thesis, Rice, 1992.

[12] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An overview of the ptran analysis system for multiprocessing," *Journal of Parallel and Distributed Computing*, vol. 5, pp. 617–640, Oct. 1988.

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

[14] M. Noakes, D. Wallach, and W. Dally, "The j-machine multicomputer: An architecture evaluation," 1993.

[15] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura, "Thread-based programming for the em-4 hybrid dataflow machine," in *Proceedings of the 19th annual international symposium on Computer architecture*, pp. 146–155, 1992.

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

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

[18] B. S. Ang, Arvind, and D. Chiou, "StarT the next generation: integrating global caches and dataflow architecture," Tech. Rep. CSG-memo-354, MIT, 1994.

[19] M. FIllo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, "The M-machine multicomputer," in *International Symposium on Computer Architecture*, 1995.

[20] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Argarwal, "Baring it all to software: Raw machines," *IEEE Computer*, 1997.

[21] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," in *Proceedings of the 30th annual international symposium on Computer architecture*, 2003.

[22] S. Sakai, y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba, "An architecture of a dataflow single chip processor," in *Proceedings of the 16th annual international symposium on Computer architecture*, pp. 46–53, ACM Press, 1989.

[23] G. M. Papadopoulos and D. E. Culler, “Monsoon: An explicit token-store architecture,” in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, (Seattle, Washington), pp. 82–91, IEEE Computer Society and ACM SIGARCH, May 28–31, 1990. *Computer Architecture News,* 18(2), June 1990.

[24] G. M. Papadopoulos and K. R. Traub, “Multithreading: A revisionist view of dataflow architectures,” in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, (Toronto, Ontario), pp. 342–351, ACM SIGARCH and IEEE Computer Society TCCA, May 27–30, 1991. *Computer Architecture News,* 19(3), May 1991.

[25] R. S. Nikhil and Arvind, “Can dataflow subsume von Neumann computing?,” in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, (Jerusalem, Israel), pp. 262–272, IEEE Computer Society TCCA and ACM SIGARCH, May 28–June 1, 1989. *Computer Architecture News,* 17(3), June 1989.

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

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

[28] A. L. Davis, “The architecure and system method of DDM1: A recursively structured data driven machine,” in *Proceedings of the 5th Annual Symposium on Computer Architecture*, (Palo Alto, California), pp. 210–215, IEEE Computer Society and ACM SIGARCH, April 3–5, 1978.

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

[30] T. Shimada, K. Hiraki, K. Nishida, and S. Sekiguchi, “Evaluation of a prototype data flow processor of the sigma-1 for scientific computations,” in *Proceedings of the 13th annual international symposium on Computer architecture*, pp. 226–234, IEEE Computer Society Press, 1986.

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

[32] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes, “The epsilon dataflow processor,” in *Proceedings of the 16th annual international symposium on Computer architecture*, pp. 36–45, ACM Press, 1989.

[33] G. Papadopoulos and D. Culler, “Monsoon: An explicit token-store architecture,” in *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990.

[34] Arvind and R. Nikhil, “Executing a program on the mit tagged-token dataflow architecture,” *IEEE Transactions on Computers*, vol. 39, no. 3, pp. 300–318, 1990.

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

[36] A. H. Veen, *The Misconstrued Semicolon: Reconciling Imperative Languages and Dataflow Machines.* Mathematish Centrum, 1980.

[37] S. Allan and A. Oldehoeft, “A flow analysis procedure for the translation of high-level languages to a data flow language,” *IEEE Transactions on Computers*, 1980.

[38] J. B. Dennis, “Dataflow supercomputers,” in *IEEE Computer*, IEEE, November 1980.

[39] “The microarchitecture of a pipelined wavescalar processor: An RTL-based study.” Submitted for blind review to MICRO-2005.

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

[41] R. L. Sites and R. L. Witek, *Alpha AXP architecture reference manual.* 12 Crosby Drive, Bedford, MA 01730, USA: Digital Press, second ed., 1995.

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

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

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

[45] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, D. M. Tullsen, and S. J. Eggers, “Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading,” *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 322–354, 1997.

[46] M. Ekman and P. Stenström, “Performance and power impact of issue-width in chip-multiprocessor cores,” in *International Conference on Paralllel Processing*, 2003.

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

[48] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, “Piranha: A scalable architecture based on single-chip multiprocessing,” in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, (Vancouver, British Columbia), pp. 282–293, IEEE Computer Society and ACM SIGARCH, June 12–14, 2000. *Computer Architecture News,* 28(2), May 2000.

[49] SPEC, “Spec CPU 2000 benchmark specifications.” SPEC2000 Benchmark Release, 2000.