Threads on the Cheap: Multithreaded Execution in a WaveCache Processor

Steven Swanson Andrew Schwerin Andrew Petersen
Mark Oskin Susan Eggers
Computer Science and Engineering
University of Washington
{swanson,schwerin,petersen,oskin,eggers}@cs.washington.edu

Abstract: *Executing multiple threads on a single processor will play a key role the future scaling of computer performance, and while many new architectures propose novel uses for threads, few address the complexity required to support multiple threads in a single processor core. This paper describes extensions to WaveScalar, a recently proposed dataflow instruction set, and the WaveCache, a WaveScalar processor, that allow multiple threads to execute simultaneously. The original WaveCache is significantly less complex than a modern out-of-order von Neumann processor, and the modifications it requires for multithreading are very small. We demonstrate that resulting multithreaded architecture can efficiently execute applications from the Splash2 benchmark suite.*

1 Introduction

Threads have proven to be an effective tool for enhancing program performance. For many years, they have supplied explicitly parallel applications, such as web servers, databases, and some scientific codes, with a simple parallel programming model and substantial speedups on multiple processors. Current research suggests that future multithreaded processors may use threads to benefit single-threaded programs by prefetching data [1], speculatively executing past branches [2] and across loop iterations [3], and improving system reliability [4, 5].

However, supporting multiple threads can introduce significant microarchitectural complexity. Simultaneous multithreading, for instance, dynamically partitions resources across one or more threads, providing good performance for both single- and multi-threaded workloads, but SMT processors are extremely complicated. They incur all the complexity costs of out-of-order execution but also must provide support for multiple threads throughout the pipeline including multiple commit queues, multiple store buffers, larger register files and renaming tables.

WaveScalar [6] is a recently developed dataflow instruction set that combines the fine-grain threading and synchronization mechanisms that the dataflow instruction firing rule provides with a simple memory ordering technique called *wave-ordered memory* [6]. This provides total-load-store ordering for coarse-grain, von Neumann-style threads. The WaveScalar instruction set runs on a new style of low-complexity, distributed processor. WaveScalar programs execute in a sea of simple processing nodes that replace the central processor and instruction cache of a conventional computer. Conceptually, instructions execute in-place in the memory system and explicitly forward their results to any dependents. In practice, they are stored and executed by an intelligent, distributed instruction cache called a *WaveCache*. The WaveCache loads instructions from memory and assigns them to processing elements for execution. The working set of the application remains in this "intelligent I-cache" over many, potentially millions of, invocations.

The chief advantages of the WaveCache over superscalars and other von Neumann processors are its low complexity; reliance on local, rather than global, communication; and high defect tolerance. Dataflow architectures are inherently multithreaded (some models even consider each instruction to be its own thread), so augmenting the WaveCache to support multiple von Neumann-style threads is much simpler and requires less hardware than adding multithreading support to an out-of-order superscalar processor.

The WaveCache consists of small, low-complexity processing nodes replicated across the die that use nearest-neighbor, point-to-point messaging. This reduces design complexity and avoids long wire-delays that can limit clock speeds. Prior work [6] demonstrates that even with the crudest of compilation tools (a binary translator to convert Alpha executables to the WaveScalar instruction set), the WaveCache outperforms an extremely aggressive superscalar design by 2-7X. This work extends the WaveCache to support multiple threads by simply widening some hardware structures by a few bits and providing additional, independent copies of others.

In this work, we describe two key aspects of the WaveScalar threading system. First, we deconstruct the prior work on WaveScalar by decoupling dataflow execution from wave-ordered memory to give threads greater control

over scarce memory-ordering hardware resources. Second, we introduce a new instruction that facilitates lightweight, *memory-less* inter-thread communication. Using this communication mechanism, we construct a variety of efficient synchronization and communication primitives including locks, barriers, condition variables, and simple message passing.

Our results demonstrate that the WaveCache can execute threaded programs efficiently. We embed our memory-less synchronization primitives in the Splash-2 benchmark suite to explore the performance and scalability of traditional coarse-grain parallel codes executing on the WaveCache.

The next section reviews the WaveScalar instruction set and the WaveCache. In Section 3, we describe extensions to the WaveScalar ISA and the WaveCache architecture to support multiple threads, decouple memory order from execution, and support efficient lightweight synchronization. Section 5 illustrates how these mechanisms are employed to build memory-less synchronization primitives. Sections 6 and 7 present an evaluation of threading in the WaveCache and our conclusions.

2 WaveScalar Review

Before discussing the architectural changes made to support threads, we provide a brief review of the WaveScalar instruction set and the WaveCache architecture. The WaveScalar ISA and its WaveCache implementation are a response to the scaling problems facing tomorrow's superscalar and VLIW processors and specifically target the increasing complexity of superscalars' centralized designs, the increasing disparity between computation and communication costs, and the decreasing reliability of shrinking circuit technology. WaveScalar side-steps these issues with a distributed computing substrate composed of thousands of simple, largely identical, and interchangeable processing elements (PEs). To reduce communication costs within this substrate, PEs are organized into clusters, each associated with its own store buffer and data cache. Instructions are placed in clusters to minimize both inter-PE and PE-memory communication.

The PEs execute instructions using a dataflow execution model [7, 8, 9, 10, 11, 12, 13, 14]. Dataflow computers execute programs according to the dataflow firing rule: instructions execute after all their operands become available. Values in a dataflow machine generally carry a tag to distinguish them from other dynamic instances of the same variable. A value and its tag are known as a token.

Previous dataflow machines have excelled at exposing parallelism but required programs to be written in special languages that eliminate side effects. WaveScalar surmounts this shortcoming with wave-ordered memory, a memory-ordering scheme that uses compiler-supplied annotations to preserve total load-store ordering and enables programs written in imperative languages to execute efficiently. In addition to language restrictions, the centralized designs of early dataflow machines' token stores hampered their performance and scalability. The WaveCache eliminates this problem by distributing the token store and matching logic across the PEs.

The consequence of these features is a dataflow execution model and implementation that are realizable in near-term technology and can efficiently execute programs written in any language.

2.1 The WaveScalar Instruction Set

Like previous dataflow instruction sets, WaveScalar converts control dependences into data dependences by sending data values to the instructions that need them. Rather than changing the value of a program counter, which causes particular instructions to be fetched and executed on a von Neumann machine, WaveScalar includes two dataflow instructions that explicitly steer values to their intended consumers. A conditional selector, ϕ [15], takes two input values and a boolean selector and, depending on the selector's value, produces one of the inputs on its output. The reverse operation, a conditional split or ϕ^{-1} [16], takes an input value and a boolean output selector and directs the input to one of two possible outputs, depending on the selector value.

Like traditional dataflow machines, WaveScalar uses tags to identify different dynamic instances of data. Unlike traditional dataflow machines, where tag creation is either partially distributed [17] or completely centralized, WaveScalar's tag control mechanism is entirely under software control, and distributed throughout the WaveCache. A special instruction, WAVE-ADVANCE, increments the tag, called WAVE-NUMBER, by 1 (modulo a maximum).

When compiling imperative language code, the WaveScalar compiler breaks the control flow graph of an application into pieces called waves. A wave's key properties are: (1) its instructions are partially ordered (there are no loops); (2) control can only enter at a single point; and (3) each time a wave executes, its instructions execute at most once. These properties allow the compiler to reason about memory ordering within a wave. Note that a wave is more general than a hyperblock [18], since it can contain joins. This makes it easy for the compiler to increase wave size by unrolling loops.

Traditional imperative languages provide a programmer with a model of memory known as total load-store ordering. WaveScalar brings this feature to dataflow computing by using *wave-ordered memory*. Under wave-ordered memory, the compiler annotates each memory operation with both its location in its wave and information about its ordering relationships, defined by the control flow graph and instruction order within basic blocks, to other memory operations in the same wave. As the memory operations execute, these annotations travel to the memory system, allowing to apply memory operations in the correct order.

Finally, WaveScalar supports object linking, shared libraries, and indirect function calls. Facilitating these con-

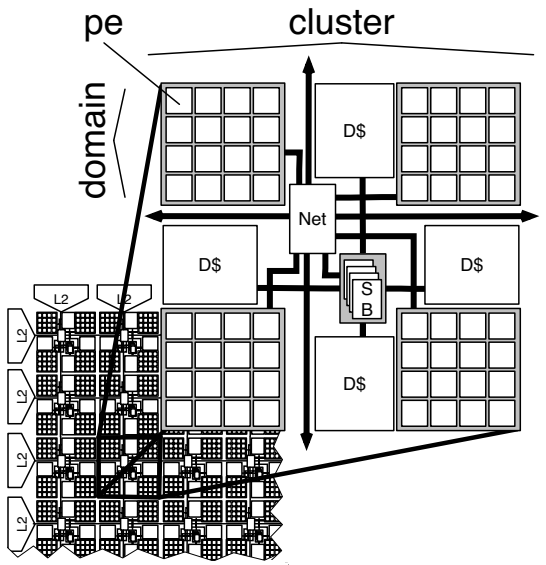

Figure 1: **WaveCache:** The WaveScalar instruction set is intended to be executed by an substrate of computational elements called the WaveCache.

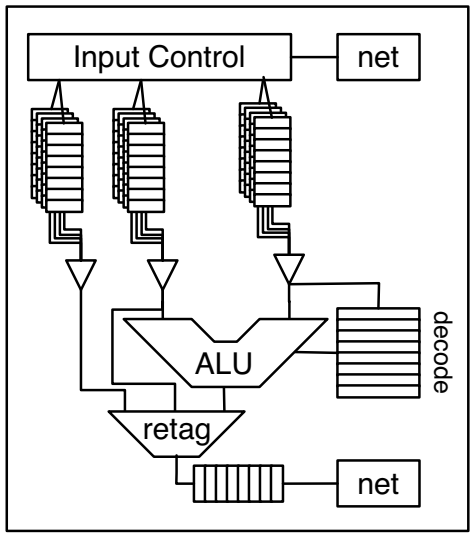

Figure 2: **WaveCache node:** Each node of the WaveCache is a simple ALU and input/output queues. Instructions are mapped onto this node which caches and executes them.

structs requires an additional instruction, INDIRECT-SEND, which takes three inputs: a data value (e.g., a function argument), an address, and an offset (which is statically encoded as an immediate value). INDIRECT-SEND sends the value to a consumer instruction located at the address plus the offset.

2.2 The WaveCache: a WaveScalar processor

In this section, we summarize the design of a WaveCache processor to execute WaveScalar binaries that could be built within the next 5-10 years (Figure 1). This microarchitecture is the baseline model used in the simulation experiments presented in later sections.

The WaveCache is a grid of simple processing elements. Each PE (Figure 2) contains buffering and storage for 8 different static instructions, although only one can fire each cycle. A PE also includes logic to control instruction placement and execution, input and output queues for instruction operands, communication logic, and a functional unit. PEs are co-located with their static instructions, forming an "intelligent" instruction cache, or processor-in-cache.

PEs are grouped into domains. Within a domain, instructions can both execute and send their results to a consuming PE within a single cycle. Four domains are grouped into a cluster, which houses both a store buffer and a traditional 4-way set-associative, 16K L1 data cache. The L1 caches are kept coherent using a simple directory-based coherence scheme that supports a single cache line owner with no sharing. There is a 16MB unified L2 cache distributed around the edge of the grid, along with the L1 directory. Once memory requests reach the L2 cache, access latencies are 20 cycles, with 1000 cycles to access main memory. The store buffer in each cluster can support four reads and writes per cycle. The on-chip network contains a 4-ported bidirectional switch at each cluster. The network latency is one cycle to arrive at the switch from a PE and then one cycle per network hop in the grid. Simulation of the system in this paper faithfully models contention on all network links and communication busses for operand, memory, and cache coherence traffic. Instruction placement is done on-demand and dynamically stripes instructions across the grid.

This design uses a distributed wave-ordered memory interface. Each dynamic wave is bound to a store buffer, which fields all memory requests for that wave. As a store buffer completes, it signals the store buffer for the next wave to proceed – analogous to a baton pass in a relay race. This scheme allows the store buffer to be logically centralized but to migrate around the WaveCache as the locus of execution moves.

3 Threads in WaveScalar

In the previous section, we described the basic WaveScalar instruction set and WaveCache microarchitecture. Here, we describe new instructions to manage multiple wave-ordered memory sequences and synchronize two threads without going through memory. Then we outline the changes required to execute multiple threads in the WaveCache.

3.1 Memory order creation and management

The WaveScalar ISA includes several instructions for managing threads. Previously we described how each dynamic value in the WaveCache is tagged with a WAVE-NUMBER. To efficiently support von Neumann-style threads, we introduce a second field, the THREAD-ID. In addition to differentiating values in different threads, the WaveCache implementation uses THREAD-IDs to replicate instructions across PEs as they are loaded.

We use a straightforward notation, $<t, w>.v$, to describe a token within the WaveScalar ISA, where t is the THREAD-ID, w is the WAVE-NUMBER, and v is the data value. To manipulate THREAD-IDs and WAVE-NUMBERs, we introduce several instructions to convert WAVE-NUMBERs, THREAD-IDs, and normal data values: DATA-TO-THREAD, THREAD-TO-DATA, DATA-TO-WAVE and WAVE-TO-DATA. These instructions fire according to the same rule as other instructions, i.e., their inputs must match on both THREAD-ID and WAVE-NUMBER.

Since each thread will requires its own total load-store ordering, we add the ability to create and manage multiple wave-ordered memory sequences. Two instructions control the creation and termination of an ordered sequence of memory operations. MEMORY-SEQUENCE-START takes two inputs: a THREAD-ID that identifies the memory sequence and a WAVE-NUMBER that is the first wave in that sequence. MEMORY-SEQUENCE-START produces the same THREAD-ID as output, signaling that the memory system is ready to receive memory requests for that sequence. A second instruction, MEMORY-SEQUENCE-STOP, terminates a memory-ordered sequence; its single input is the THREAD-ID of the memory sequence to terminate. The memory system treats MEMORY-SEQUENCE-STOP similarly to an ordered store instruction, ensuring that all previous memory operations in the sequence have finished before the sequence terminates

To create an ordered thread, a program uses MEMORY-SEQUENCE-START to turn on memory ordering for a given THREAD-ID, and then DATA-TO-THREAD and DATA-TO-WAVE to set the THREAD-ID and WAVE-NUMBER on the appropriate instructions. When the thread finishes, it calls MEMORY-SEQUENCE-STOP to release the state associated with its memory-ordered operations.

3.2 Memory-less synchronization

The final instruction, THREAD-COORDINATE, allows computations executing under different THREAD-IDs to communicate directly, rather than through shared memory. THREAD-COORDINATE has unique semantics in the WaveScalar ISA, since it uses a modified firing rule. A THREAD-COORDINATE instruction fires when the data value on its first (TAKE) input matches the THREAD-ID of a value on its second (PUT) input. THREAD-COORDINATE produces an output with the THREAD-ID and WAVE-NUMBER from the TAKE input and the data value from the PUT input. For example, if $<T, w>.K$ is the TAKE input and $<K, u>.v$ is the PUT input, THREAD-COORDINATE will produce the output, $<T, w>.v$. To prevent starvation, THREAD-COORDINATE must consume values in the order in which they arrive.

THREAD-COORDINATE is similar in spirit to other lightweight synchronization primitives [19, 20], but is tailored to WaveScalar's dataflow framework. We will demonstrate in Section 5 how to use THREAD-COORDINATE to create lightweight memory-free locks and barriers that can coordinate von Neumann-style threads running in the WaveCache.

4 Threads in the WaveCache

The original WaveCache architecture was designed to execute a single coarse-grained. Adding support for multiple, high-performance threads requires small changes to the design of the processing elements, the store buffers, the communication infrastructure and the instruction placement controller.

The principle change is the additional bus width required to transmit the extended tag (THREAD-ID and WAVE-NUMBER). To support multiple threads in the PE input queues, we can simply added a small register to hold the current THREAD-ID for all data values in the input queues, since the queues are not shared across threads.

The placement algorithm now dynamically loads a copy of an instruction for each THREAD-ID that uses it, allowing the number of static instances of an instruction to grow the number of threads that execute that instruction and reducing contention for each instance.

The only change required to the store buffers is the additional hardware to maintain the memory sequences of multiple wave-ordered threads. Like the input queues at the PEs, the store buffers are not shared across threads, so a register holds the THREAD-ID and WAVE-NUMBER for the thread currently using the store buffer.

5 Inter-thread synchronization

Traditional threading systems provide a set of function calls and data types that support thread management and synchronization. On most coarse-grained architectures, these mechanisms are heavyweight, because the hardware only provides synchronization primitives that act through memory. WaveScalar sidesteps main memory for all synchronization, and uses the THREAD-COORDINATE instruction to synchronize threads (Sections 5.1 through 5.3).

5.1 General mutexes

Recall from Section 3.2 that the THREAD-COORDINATE instruction allows two threads to pass a data value. By varying the meaning of that value, one can create a variety of

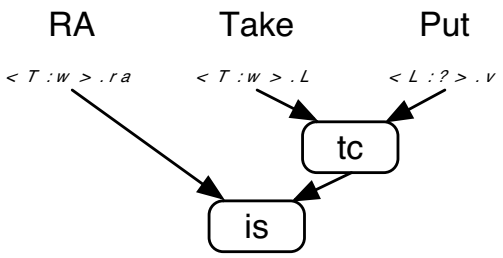

Figure 3: A memory-less mutex.

synchronization objects familiar to imperative-language programmers and essential to operating systems. We discuss two of these, *mutexes* and *barriers*, along with the general thread communication mechanism that underpins them.

Combining THREAD-COORDINATE with an INDIRECT-SEND instruction, as shown in Figure 3, forms a simple mutex that does not rely on the memory system. Instead, threads acquire a mutex by consuming a *lock token* and release it by returning the token.

To acquire a mutex L, a thread T on wave w sends a token $<T, w>.L$ to the TAKE port of the mutex's THREAD-COORDINATE instruction, and $<T, w>.ra$, where ra is the address of the instruction that should receive the lock token, to the address port of the mutex's INDIRECT-SEND. If L is unlocked, a lock token with THREAD-ID L will be sitting in an input queue of THREAD-COORDINATE's PUT port. THREAD-COORDINATE can then fire, sending its result, $<T, w>.v$, to the mutex's INDIRECT-SEND instruction. Once the INDIRECT-SEND receives ra and the lock token, it forwards the token to the instruction located at ra. That instruction can then trigger its dependents, returning control to the locking thread. However, if a token with THREAD-ID L is *not* present on the PUT port of the THREAD-COORDINATE instruction, L is locked, and the THREAD-COORDINATE cannot fire until L's current holder releases it. Therefore, the instructions dependent on the instruction at ra cannot fire.

A thread releases a mutex by sending the lock token to the THREAD-COORDINATE's PUT port. To guarantee that all memory operations in a critical section protected by the mutex complete before it releases the mutex, a thread holding the mutex issues a MEMORY-NOP-ACK instruction, which stalls until all previous ordered memory operations have finished. Finally, it makes the mutex release dependent on the value returned by MEMORY-NOP-ACK.

5.2 Inter-thread communication using THREAD-COORDINATE

The THREAD-COORDINATE instruction also provides a mechanism for arbitrary inter-thread communication without accessing the memory system. For instance, let C be a (client) thread that wishes to initiate communication with another (server) thread, S. Using a THREAD-COORDINATE instruction, S publishes a wave number u for which it will receive values (by making u the data value of a PUT token), essentially creating a "your number is" ticket that can be used for later communications. Thread C in wave w, takes the next available ticket by sending $<C, w>.S$ to THREAD-COORDINATE's TAKE port, receiving wave number u. It can then send messages to thread S for wave u. In order to receive a response from S, C also includes its current wave number w along with the other messages.

Although there may be any number of client threads, arriving in any order, this implementation allows only a single server thread. A more general solution to the problem of matching data producers to data consumers would have the single server thread manage a pool of worker threads to act as a matchmaker between clients and available servers.

Algorithm 1 barrier_thread(*max_count*)

Require: *max_count* is the desired release threshold
  tcount $\leftarrow 0$
  next_link $\leftarrow$ NIL
  loop
    receive message "barrier_wait(*requestor*)"
    send message "chain_link(*next_link*)" **to** *requestor*
    next_link $\leftarrow$ *requestor*
    tcount $\leftarrow$ *tcount* $+ 1$
    if *tcount* $=$ *max_count* **then**
      send message "barrier_release(*tcount* $- 1$)" **to** *requestor*
      tcount $\leftarrow 0$
    end if
  end loop

5.3 General purpose barriers

A barrier is a specialized condition variable commonly used in thread synchronization. It supports one important operation: *wait*. Calls to *wait* block until some specified number of threads have arrived at the barrier. After this condition is met, all waiting threads are released, i.e., *wait* returns in each of the waiting threads, and their execution continues.

Our implementation of a general purpose barrier uses the inter-thread communication method described above and no memory operations. A barrier object is simply a thread executing the loop in Algorithm 1. A thread waits on a barrier by sending it a *wait* message and making its subsequent operations dependent on the barrier's reply. The barrier responds immediately with a *chain-link* message, notifying a waiting thread of the THREAD-ID and WAVE-NUMBER of the previous waiter (if there is one). When the exit condition is met, the barrier sends a *release* message to the last thread to arrive. Upon receiving a *release*, a thread propagates it to the thread

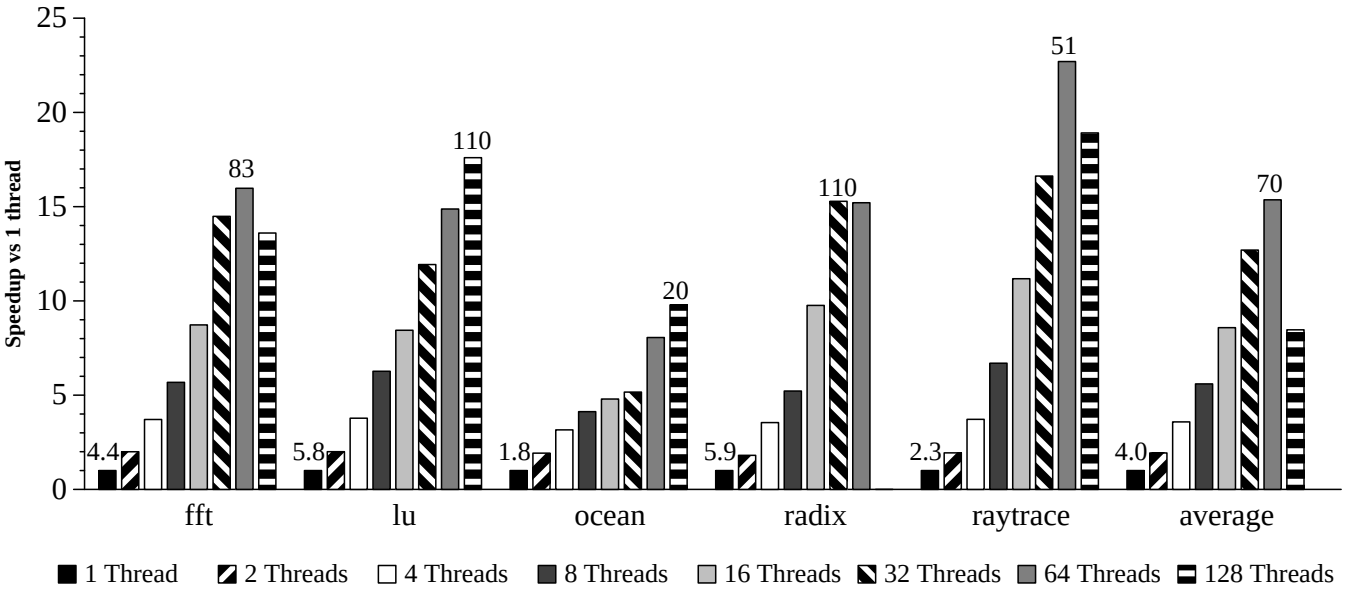

Figure 4: **Splash-2 on the WaveCache.** We evaluate each of our Splash-2 benchmarks on the baseline WaveCache with between 1 and 128 threads. The bars represent speedup in total execution time. The numbers above the single-threaded bars are IPC for that configuration. One benchmark, *radix*, cannot utilize 128 threads with the input data set we use, so that value is absent.

identified in its *chain-link* message. Distributing the queue of waiting threads across the waiting threads is a convenient way to eliminate dependence on the memory system. A straightforward extension of this barrier design provides a general condition variable. We omit its description for brevity.

6 Results

In this section, we evaluate the WaveCache with five applications from the Splash-2 benchmark suite, each modified to use the memory-less synchronization primitives described in Section 5. We compiled each application with the DEC cc compiler using -O4 optimizations. A 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 2.2.

Figure 4 contains speedup results for all five benchmarks, as compared to their single-threaded running time. On average, the WaveCache achieves over 11× speedup with 16 threads and 15× speedup with 64 threads, although *raytrace* reaches a substantially better 22×.

Increasing to 128 threads reduces performance (except for *lu* and *ocean*), because the WaveCache becomes congested by the larger instruction working set and L1 data evictions due to capacity misses. Note, however, that unlike Von Neumann multiprocessors, the performance improvement gained by adding additional threads comes with no additional hardware. That an application achieves better performance at 64 threads than 128 simply means that it fully utilizes the WaveCache's resources with fewer threads and, therefore, should be run with that number.

7 Conclusion

We have presented simple extensions to the WaveScalar instruction set that facilitate execution of multiple threads. The WaveCache architecture requires only very minor extensions to support multiple threads, and the resulting architecture is much simpler than a traditional, von Neumann-based SMT processor. Using these modest extensions to the ISA and hardware combined with our memory-less synchronization primitives, the WaveCache can effeciently execute multithreaded programs written in traditional imperative languages, such as those from the Splash-2 benchmark suite.

References

[1] A. Roth, A. Moshovos, and G. S. Sohi, "Dependence based prefetching for linked data structures," in *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pp. 115–126, ACM Press, 1998.

[2] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, (Vancouver, British Columbia), pp. 1–12, IEEE Computer Society and ACM SIGARCH, June 12–14, 2000. *Computer Architecture News*, 28(2), May 2000.

[3] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proceedings of the 22nd annual international symposium on Computer architecture*, ACM Press, 1995.

[4] E. Rotenberg, "Ar-smt: A microarchitectural approach to fault tolerance in microprocessors," in *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, June 1999.

[5] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for chip multiprocessors," in *Proceedings of the 30th annual international symposium on Computer architecture*, pp. 98–109, ACM Press, 2003.

[6] S. Swanson and M. Oskin, "Wavescalar," in *International Symposium on Microarchitecture*, 2003.

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

[8] A. L. Davis, "The architecture and system method of ddm1: A recursively structured data driven machine," in *Proceedings of the fifth annual symposium on Computer architecture*, 1978.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, October 1991.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.