WaveScalar

Steven Swanson, Ken Michelson and Mark Oskin
Deptartment of Computer Science and Engineering
University of Washington

Technical Report UW-CSE-03-01-01

January 23rd, 2003

Abstract

Silicon technology will continue to provide an exponential increase in the availability of raw transistors. Effectively translating this resource into application performance, however, is an open challenge. Ever increasing wire-delay relative to switching speed and the exponential cost of circuit complexity make simply scaling up existing processor designs futile. In this paper, we present an alternative to superscalar design, WaveScalar.

WaveScalar is a dataflow instruction set architecture and execution model designed for the construction of intelligent computation caches. Each instruction in a WaveScalar binary executes in place in the memory system and explicitly communicates with its dependents in dataflow fashion. WaveScalar architectures cache instructions and the values they operate on together in a WaveCache, a simple grid of "processor-in-cache" nodes. By co-locating computation and data in physical space, the WaveCache minimizes long wire, high-latency communication. This paper introduces the WaveScalar instruction set and an implementation based on current technology, which is evaluated for its performance potential. Results of the SPEC and mediabench applications demonstrate a factor of 2-4 performance improvement compared to an aggressively configured superscalar design.

1 Introduction

It is widely accepted that Moore's Law growth in available transistors will continue for the foreseeable future. Recent research [1], however, has demonstrated that simply scaling up our current architectures will not convert these new transistors to commensurate increases in performance. This gap, between the performance improvements we need and those we can achieve by simply constructing larger versions of existing architectures, will fundamentally alter processor designs.

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

Like the memory wall, the processor scaling wall has motivated a number of research efforts [2, 3, 4, 5]. These efforts all augment the existing Von-Neumann model of computation by providing redundant checking mechanisms [2], by exploiting compiler technology for limited dataflow-like execution [3], or by efficiently exploiting coarse-grained parallelism [5, 4]. In this paper we propose another approach, WaveScalar, that does not rely upon traditional Von-Neumann designs.

At its core, WaveScalar is a dataflow instruction set and computing model [6], but unlike past dataflow work which focused on maximizing processor utilization, WaveScalar's goal is to rid the processor of long

wires and broadcast networks and so minimize communication costs. To this end, it includes a completely decentralized scheme for tag management and matching. Uniquely, WaveScalar efficiently supports traditional Von-Neumann-like memory semantics in a dataflow model. This allows it to execute applications written in conventional languages like C, C++ and Java. Indeed, for our performance studies (Section 4) we use a binary re-writer that translates programs from the Alpha ISA to the WaveScalar instruction set.

WaveScalar is designed for intelligent cache-only computing systems. A cache-only computing architecture has no central processing unit, but rather consists of a sea of processing nodes in a substrate that effectively replaces the central processor and instruction cache of a conventional system. Conceptually, WaveScalar instructions execute in-place in the memory system and explicitly send their results to their dependents. In practice, WaveScalar instructions are cached and executed by an intelligent, distributed instruction cache – the *WaveCache*.

The WaveCache loads instructions from memory and assigns them to processing elements for execution. They remain in the cache over many, potentially millions, of invocations. Remaining in the cache for long periods of time enables dynamic optimization of an instruction's physical placement in relation to its dependents. Optimizing instruction placement allows a WaveCache to take advantage of predictability in the dynamic data dependencies of a program, which we call *dataflow locality*. Just like conventional forms of locality (temporal and spatial), dataflow locality can be exploited by the cache-like structure of the WaveCache.

This paper is intended to be the first in a sequence of studies on WaveScalar architectures. It makes four principle contributions:

1. A dataflow instruction set, WaveScalar, that includes a novel memory ordering model, *wave-ordered memory*, to provide traditional memory semantics and allow execution of programs written in conventional, imperative programming languages.

2. An efficient dataflow tag management scheme that is fully distributed and under compiler control.

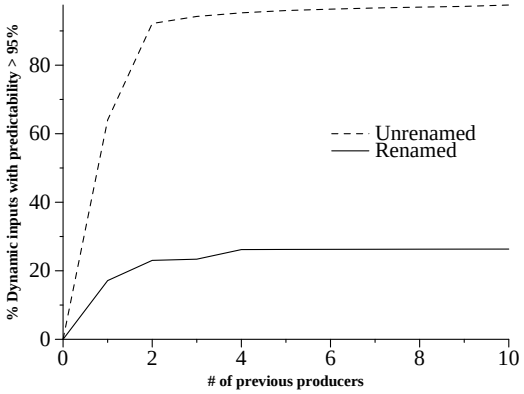
3. An implementation, the WaveCache, that could be built using current technology. The WaveScalar ISA and the WaveCache are both built to exploit dataflow locality.

4. A performance study, using the SPECint, SPECfp and mediabench benchmarks that compares the WaveCache design to an aggressive out-of-order superscalar and demonstrates the WaveCache's potential to achieve significant performance gains.

We motivate the WaveScalar model with an examination of three key unsolved challenges with superscalar designs in Section 2. In Section 3, we describe the WaveScalar instruction set and the WaveCache design. Section 4 presents an initial evaluation of our WaveCache design, and Section 5 discusses related work in this area. Finally in Sections 6 and 7, we outline directions for future WaveScalar research and conclude.

2 A case for exploring superscalar alternatives

The Von Neumann model of execution and its most sophisticated implementations, out-of-order superscalars, have been a phenomenal success. However, superscalars suffer from several drawbacks that are beginning to emerge: their inherent complexity makes efficient implementation a daunting challenge, they ignore an important source of locality in instruction streams, and their execution model centers around instruction fetch, an intrinsic serialization point. We examine each of these limitations in turn.



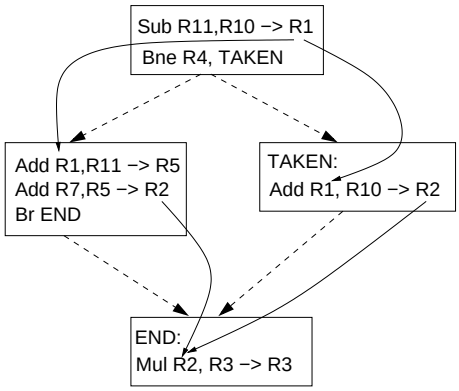


Figure 1: In this figure (left), we quantify the amount of dataflow locality present in the SPEC2000 integer benchmark suite. The y-axis is the percent of instruction inputs for which the source instructions is one of the last unique n (x-axis) producers for that input. This quantity corresponds to the dataflow locality present in the instruction stream. For instance, without renaming, 92% of values come from 1 of the previous 2 sources. With register renaming, this number drops to 23%, clearly demonstrating the detrimental effects of register renaming on dataflow locality. On the right, we show a simple example of how dataflow locality arises. Without renaming, the `Mul` instruction always gets its input from one of the `Add`'s via `R1`. With renaming, however, the value can come via any physical register.

2.1 Manufacturing and design

As features and cycle times shrink, the hardware structures that form the core of superscalar processors (register files, issue windows, scheduling logic and caches) become extremely expensive to access. Consequently, clock speed and/or pipeline depth suffers. Indeed, industry recognizes that scaling superscalar designs with transistor budgets can be impractical, and many manufacturers are turning to larger caches and chip multiprocessors for increased performance.

Second, the complexity of modern processors means that verification is an ever increasing cost in processor design. To squeeze maximum performance from the core, more complex algorithms and structures are required. Each new mechanism, optimization, or predictor adds additional complexity and increases verification time. Already, design verification consumes 40% of project resources on complex designs [7] and verification costs are increasing.

Finally, most traditional designs are not tolerant of manufacturing defects. Manufacturers are beginning to address this problem. The McKinley (Itanium 2) and upcoming Madison processors from Intel contain a good deal of redundancy to increase yields [8]. Eventually, however, it will be necessary to design processors to be fault tolerant from day one [2].

2.2 Untapped locality

Superscalars devote a large share of their hardware and complexity to exploiting locality and predictability in program behavior. However, they fail to utilize a significant source of locality intrinsic to applications: *dataflow locality*. Dataflow locality is the predictability of instruction dependencies through the dynamic trace of an application.

Dataflow locality exists, because instructions only communicate with a few other static instructions in a program binary, and they communicate via fixed registers encoded in the instructions. Although, there may be several potential producers of an input to a given static instruction along different control paths, our

measurements show that 95% of instruction input values have only 1 or 2 potential static producers. Figure 1 illustrates this effect by measuring dataflow locality for the Spec2000 benchmarks.

Instead of simply ignoring dataflow locality, however, superscalars destroy it in their search for parallelism. Fetching the same static instructions repeatedly, such as within a loop, causes the same register name to be used repeatedly, leading to both write-after-write (WAW) and write-after-read (WAR) conflicts. Register renaming [9, 10] removes these false dependencies and allows different physical registers to correspond to different dynamic instances of the same architectural register. Renaming enables dynamic loop unrolling and exposes a large amount of dynamic ILP for the superscalar core to exploit.

However, register renaming destroys the predictability of data communication patterns. Without renaming, each instruction would only need high-speed access to (at most) three registers, since its source and destination registers are fixed. By changing the physical registers an instruction uses, renaming forces the architecture to provide each instruction with fast access to the entire physical register file. The result is a huge, slow register file and complicated forwarding networks.

Destroying dataflow locality leads to a shocking inefficiency in modern processor designs: The processor fetches a stream of instructions with a highly predictable (upwards of 90%) point-to-point (i.e., instruction-to-instruction) communication pattern, destroys that predictability (to well below 30%) by renaming, and then compensates by using broadcast communication in the register file and a by-pass network combined with complex scheduling in the instruction queue. The consequence is that modern processor designs devote few resources to actual execution (less than 10%, as measured on a Pentium III die photo), and the vast majority to communication infrastructure. This infrastructure is necessary precisely because superscalars do not exploit dataflow locality.

It is an open question whether a Von-Neumann machine can ever effectively exploit this type of locality. Aside from a few research proposals [3, 11, 12], modern processors have not tried to aggressively exploit dataflow locality. Partitioned superscalars like the Alpha 21264 and some VLIW machines [13, 14] exploit it to a limited degree, but neither is able to make full use of it. The GPA project [3] suggests that it may be possible to better utilize dataflow locality, although further study is needed. In Section 3, we present an execution model and architecture built expressly to exploit the temporal, spatial, and dataflow locality that exist in instruction and data streams.

2.3 The Von Neumann model: serial computing

In addition to design difficulties and unexploited locality, superscalars (and Von Neumann machines in general) suffer from a more subtle, but fundamental, limitation. In the Von Neumann model, the processor, guided by the program counter and control instructions, assembles a linear sequence of operations for execution. The elegance and the simplicity of the model are striking, but the price is steep. Von Neumann processors are fundamentally sequential. There is no parallelism in the model.

In practice, of course, Von Neumann processors do achieve limited parallelism (e.g., IPCs greater than 1), by using one of two methods. The explicitly parallel instructions sets for VLIW and vector machines enable the compiler to express instruction independence statically. Superscalars take a different approach and examine many instructions in the execution stream simultaneously, violating the sequential ordering when they determine it is safe to do so.

Prior work [15, 16, 17, 18] demonstrated that ample instruction level parallelism (ILP) exists within applications, but the control dependencies that sequential fetch introduces constrains this ILP. Despite tremendous effort over decades of computer architecture research, we have yet to devise a processor that comes close to the intrinsic ILP limits researchers have measured in limit studies. Several factors account for this, including the memory wall and necessarily finite execution resources, but control dependence and, by extension, the inherently sequential nature of Von Neumann execution, remain dominant factors [15]. In the next section, we describe an alternative model that does not introduce false control dependencies.

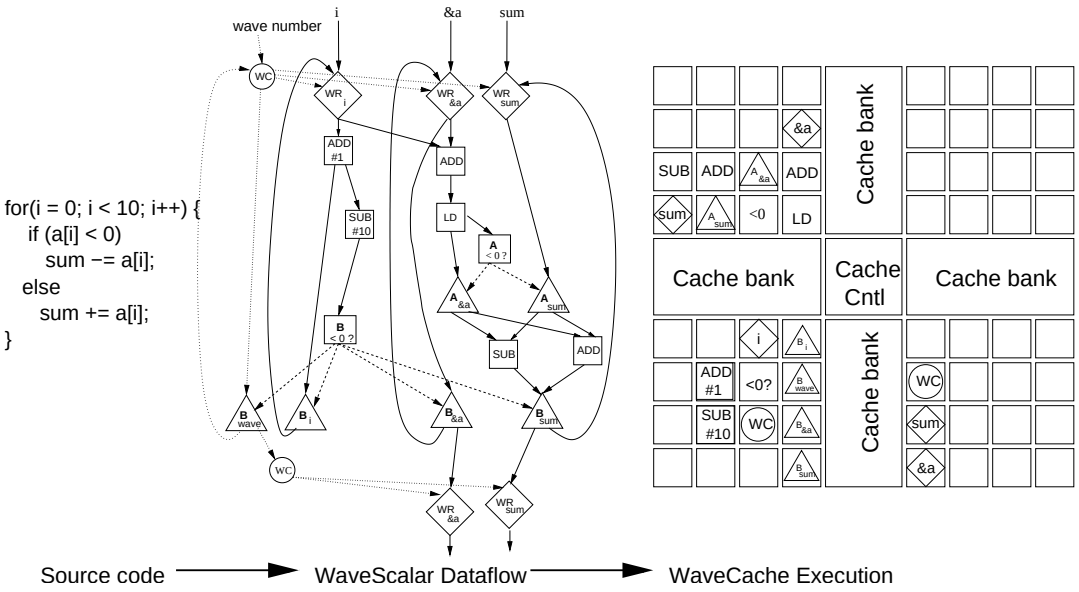


Figure 2: **Wavescalar execution**: At left is a loop with some simple control and two loop-carried dependencies. The center images show the WaveScalar "binary" for the code. In addition to RISC instructions, it includes WAVE-COORDINATE (circles), WAVE-RENAME (diamonds), and several ϕ^{-1} instructions (triangles). The two groups of a branch instruction and several ϕ^{-1} instructions, labeled A and B, correspond to the branches in- and outside the loop, respectively. The loop body comprises a single wave with a WAVE-COORDINATE and three WAVE-RENAME instructions to rename each value at the top of the loop. Another WAVE-COORDINATE and two more WAVE-RENAME instructions set the wave number for the code that follows the loop (not shown). At right, each of the instructions is attached to a functional unit in the WaveCache substrate.

3 WaveScalar

The original motivation for WaveScalar was to build a decentralized superscalar processor core, not to create a dataflow architecture. Our initial approach was to examine each piece of a superscalar and try to design a new, decentralized hardware algorithm for it. Our thesis was that by decentralizing everything, we could design a truly scalable superscalar. It soon became apparent that instruction fetch is difficult to decentralize, because, by its very nature, a single program counter controls it. Our response was to make the processor fetch in data-driven rather than program counter-driven order. From there, our "superscalar" processor quickly became a small dataflow machine, and other parallels began to appear. Register renaming in a Von-Neumann machine corresponds to tag creation in a dataflow processor, the issue window corresponds to the token store, and the scheduling and execution rules are essentially the same (i.e., instructions fire when their inputs are ready). The problem then became how to build a fully decentralized dataflow machine. WaveScalar is the creative extension of this line of reasoning.

Dataflow has a long history. The first designs appeared in the early 70's [6, 19, 20], and there was a significant revival in the 80's and early 90's [21, 22, 23, 24, 25, 26, 27]. Dataflow machines execute programs according to the dataflow firing rule (DFR), which stipulates that an instruction may execute at any time, as long as its operands are available. When dataflow instructions complete, they trigger the execution of dependent instructions. Values in a dataflow machine generally carry a tag to distinguish them from other dynamic instances of the same variable. Tagged values usually reside in a specialized memory (the token store) while waiting for an instruction to consume them. There are, of course, many variations on

this basic dataflow idea. We discuss some of these in Section 5.

We propose a new approach to dataflow computing, *WaveScalar*, that addresses the problems discussed in Section 2. WaveScalar's primary goal is to exploit dataflow locality. Since it is a dataflow instruction set, it avoids the serialization inherent in the fetch stage of the Von Neumann model. However, it also provides two significant features absent in traditional dataflow designs: The ability to efficiently execute programs written in traditional imperative languages, such as C, C++ and Java, and completely distributed tag management.

Conceptually a WaveScalar binary is the dataflow graph of an executable and resides in memory as a collection of *intelligent* instruction words. Each instruction word is intelligent because it has a dedicated functional unit. Since this is impractical, instructions are cached by an intelligent instruction cache, a *WaveCache*, in practice.

Figure 2 illustrates the relationships between C source code, a WaveScalar binary, and a WaveCache processor. The left portion of the figure is a simple loop containing an if-then-else construct. The WaveScalar compiler produces the data flow graph for the loop (middle). The WaveScalar version of the loop contains special instructions to ensure that all control and data dependencies between instructions are explicit. The added instructions will be discussed shortly.

In Figure 2 (right), we see each instruction bound to an intelligent cache "line" (a simple functional unit). After the WaveCache assigns them to a processing element, instructions remain in place for possibly millions of executions. Instructions exchange data values over a switched, dynamically scheduled on-chip network. Communication latency depends on instruction placement, so a good placement is paramount for optimal performance.

In the next two sections, we use this example to motivate our description of the WaveScalar ISA and a sample WaveCache design that could be built today.

3.1 The WaveScalar ISA

A WaveScalar executable contains an encoding of the program dataflow graph. In addition to normal RISC-like instructions, WaveScalar provides special instructions for managing control flow. In this respect WaveScalar is similar to previous dataflow assembly languages[20, 28, 27]. WaveScalar is unique, however, because it includes a mechanism for expressing independence among memory operations and support for distributed tag management. Also unlike all previous dataflow work that we are aware of, WaveScalar targets programs written in mainstream imperative languages (such as C), instead of those written in specialized dataflow languages [29, 30, 31, 32, 33, 34, 35]. There have been some prior attempts at this [36, 37]. However, none adequately addressed the most difficult challenges including pointers, aliasing, and mutable data structures. The specific differences between the WaveScalar ISA and a normal RISC-like instruction set are described below.

3.1.1 Control flow within a DAG

A WaveScalar instruction must explicitly send data to its consumers instead of simply storing values in the register file to be read by instructions that need them. Within a basic block, this can be done simply by addressing the dependent instruction directly. In general, however, values need to cross basic blocks boundaries. The potential consumers are known at compile time, but depending on control flow, only a subset of them should receive the values at run-time. There are two solutions to this problem, and different dataflow ISAs have used one or both.

The first solution is a conditional selector, or ϕ, instruction [38]. These instructions take two input values and a boolean selector input and, depending on the selector, produce one of the inputs on their output. ϕ instructions are analogous to conditional moves and provide a form of predication. They are desirable

because they remove the selector input from the critical path of some computations and therefore increase parallelism. They are also somewhat wasteful because they discard the unselected input.

The alternative is a conditional split, or ϕ^{-1} [27] instruction, the opposite of a ϕ instruction. The ϕ^{-1} instructions take an input value and a boolean output selector. It directs the input to one of two possible outputs depending on the selector value, effectively steering data values to the instructions that should receive them. These instructions correspond most directly with traditional branch instructions, and they are required for implementing loops.

In the example of WaveScalar code in Figure 2 the triangular nodes are ϕ^{-1} instructions. There are two sets of ϕ^{-1} instructions, labeled A and B. Set A guides data through the IF-THEN-ELSE inside the loop, and group B manages loop control. Each set receive its selector value via a set of dotted lines from the predicate instruction with the same label. Each set of ϕ^{-1} instructions, combined with its predicate, corresponds to a branch instruction in a conventional RISC assembly language. Note that set A contains only two ϕ^{-1} instructions, because the conditional instructions only use a and *sum*. In contrast, group B requires four: one each for i, a, *sum*, and the wave number (See below).

3.1.2 Wave numbers

A significant source of complexity in WaveScalar is that instructions can operate on several instances of data simultaneously. For instance, consider a loop. A traditional out-of-order machine can execute multiple iterations simultaneously, because it creates a copy of each instruction for each iteration. In WaveScalar, the same processing element handles the instruction for all iterations. Therefore, some disambiguation must occur to ensure that the instruction operates on values from one iteration at a time.

Just as a superscalar uses renamed registers, a traditional dataflow machine uses *tags* to identify different dynamic instances. In WaveScalar, we aggregate tag management across a directed acyclic graph (DAG) of basic blocks called a *wave*. *Wave numbers* differentiate between dynamic waves. Two special types of instructions manage wave numbers:

- **WAVE-COORDINATE:** The WAVE-COORDINATE instruction takes as input an existing wave number, increments it (modulo a maximum), and sends the new value to the associated WAVE-RENAME instruction(s) and the following WAVE-COORDINATE instruction via one or more ϕ^{-1} or ϕ instructions.
- **WAVE-RENAME:** The WAVE-RENAME instruction takes as input a data value and a wave number. It replaces the wave number of the data value with the new wave number it receives from the WAVE-COORDINATE instruction.

To use these instructions, the compiler (or binary translator in our case) begins by partitioning an application's control flow graph into waves. Each wave is a connected, directed acyclic graph with a single entrance and the additional constraint that, for every node, either all or none of its predecessors may be in the same wave. We partition an application into maximal waves and add a single WAVE-COORDINATE node and one WAVE-RENAME node for each of the wave's live input values. These nodes reside at the entrance of the wave and ensure that all data values entering the wave have the same wave number.

In our example (Figure 2), WAVE-COORDINATE instructions are circles and WAVE-RENAME instructions are diamond-shaped. A simple inner loop's body comprises a single wave and requires two WAVE-COORDINATE instructions, one for the loop body and one for the follow-on code. Each WAVE-COORDINATE sends a new wave number to its set of WAVE-RENAME instructions. In the example, the loop body requires three WAVE-RENAME's for the three live values in the loop body (i, a, and *sum*), while follow-on code requires only two since the value i is not used outside the loop.

A key feature of WaveScalar is that the WAVE-COORDINATE and WAVE-RENAME instructions allow wave-number management to be entirely distributed and under software control. This is in contrast to

traditional dataflow machines in which tag creation is either partially distributed or completely centralized [30].

Our system for managing wave numbers has two other advantages. First, since they can contain control flow joins, waves are more general than hyperblocks [39]. Allowing joins enables a compiler to increase a wave's size and expose additional parallelism with simple techniques such as loop unrolling. Since at least one WAVE-RENAME instruction in each wave is always on the critical path (they must relabel all incoming values) but only appear at the beginning of a wave, larger waves are desirable.

3.1.3 Indirect jumps

Modern systems rely upon object linking and shared libraries, and many codes rely upon indirect function calls. Supporting these constructs requires an additional instruction:

- **INDIRECT-SEND:** INDIRECT-SEND instructions have three inputs: a data value (i.e. a function argument), an address, and an offset (which is statically encoded into the instruction). It sends the value to the consumer instruction located at the address plus the offset.

Using this instruction, we can call a function and return values. Each argument to the function is passed through its own INDIRECT-SEND instruction. At the start of a function, a set of instructions receives these operands and starts the function execution. The function address need not be known at compile time, and a very similar mechanism allows for indirect jumps.

3.1.4 Memory ordering

Traditional imperative languages provide the programmer with a model of memory known as "total load-store ordering." Coupled with indirect addressing and memory aliasing, they leave the hardware little room to maneuver when it comes to extracting parallelism from memory accesses.

A naive method of supporting total load-store ordering in a dataflow machine is to pass a value from one memory operation to the next; each operation increments the value and includes it with its request to memory. A store buffer can then reconstruct the load-store order and process the memory operations correctly. The problem with this approach is that the ordering value becomes dependent on the control flow of a program. Since nearly every basic block contains a memory operation, the token becomes dependent on the outcome of nearly every branch. Consequently, basic blocks that could execute in parallel in the WaveCache would execute serially.

For WaveScalar, we developed a far more efficient method, *wave-ordered memory*. The WaveScalar compiler statically assigns a unique (within a wave) sequence number to each memory operation in breadth first fashion, ensuring that sequence numbers increase along any path through the wave. Next, it labels each memory operation with the sequence numbers of the predecessor and successor memory operations, if they can be uniquely determined. Because of branches and joins, there can be multiple predecessor or successor memory operations. In these cases, the compiler uses a special wild-card value, '?', instead. The combination of an instruction's sequence number and the predecessor and successor sequence numbers form a *link*, which we denote $< \mathrm{pred}, \mathrm{this}, \mathrm{succ} >$.

When a load or store instruction executes, it sends its link, its wave number (taken from an input value), an address, and data (for a store) to the memory. The memory system uses this information to assemble the correct sequence of loads and stores. This is possible because a memory instruction's link and wave number provide a total ordering on memory operations through any traversal of a wave, and, by extension, an application. To guarantee a total ordering, no path through the program may contain a pair of memory operations in which the first operation's *succ* value and the second operation's *pred* value are both '?'. If such a situation occurs, the compiler adds a special MEMORY-NOP instructions to remove the ambiguity.

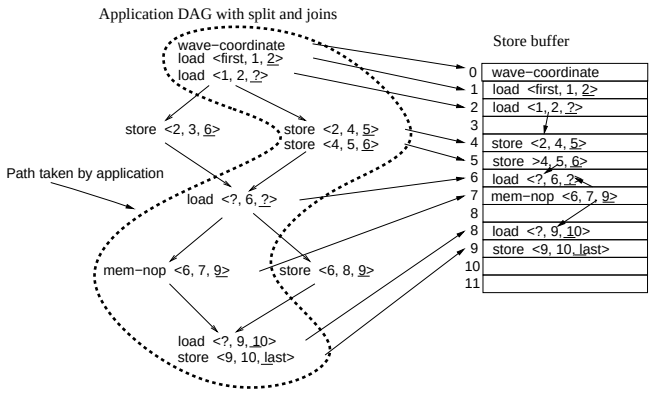


Figure 3: Memory operations are ordered through a combination of static sequence numbers and dynamic wave numbers. The key is that through any path taken through a wave, complete knowledge about the load store order is available.

These instructions participate in memory ordering but have no effect. In practice, MEMORY-NOP's are rare (less than 3% of instruction).

Figure 3 provides an example of wave-ordered memory in action. For any two consecutive memory accesses, A and B, either the A's 'succ' value matches B's sequence number or B's 'pred' value matches A's sequence number. In a WaveScalar binary, this property holds on all possible paths through a wave. Furthermore, the store buffer detects gaps in the sequence of operations and waits for the necessary operation to arrive.

Wave-ordered memory is the key to efficiently executing programs written in conventional languages. It allows WaveScalar to separate memory ordering from control flow. The processing elements are freed from worrying about implicit dependencies through memory and can treat memory operations just like other instructions. The sequencing information included with memory requests provides a concise summary of the path taken through the program. The memory systems can use this summary in a variety of ways. Figure 3 depicts the operation of a wave-ordered store buffer. Alternatively, a speculative memory system [40, 41, 42] could use the ordering to detect misspeculations.

In the future, we plan to extend wave-ordered memory to allow for multiple, parallel streams of memory operations. This will allow the compiler to describe dependence and independence relationships explicitly.

3.1.5 WaveScalar overhead and encoding

Converting from a RISC instruction set to WaveScalar increases code size in two ways. First, the individual instructions require more than 32 bits to encode because WaveScalar uses a target-based encoding scheme. Second, it is clear from the previous discussion that WaveScalar executables contain more instructions than their RISC counterparts.

The number of bits in an encoded instruction is not a significant factor since the WaveCache loads instructions infrequently and can amortize the cost over many invocations.

Instruction count, however, is more important because there is limited space in the WaveCache. To reduce overhead, the compiler can fold ϕ^{-1} and WAVE-RENAME instructions into the preceding arithmetic or logical operation, because they are so simple. For instance, the WaveScalar ISA contains both a normal ADD and a ADD-WITH-WAVE-RENAME instruction. This might modestly increase the number of bits in each instruction, but it also dramatically decreases the total instruction count.

For the applications we consider in Section 4, folding decreases instruction count overhead by up to a

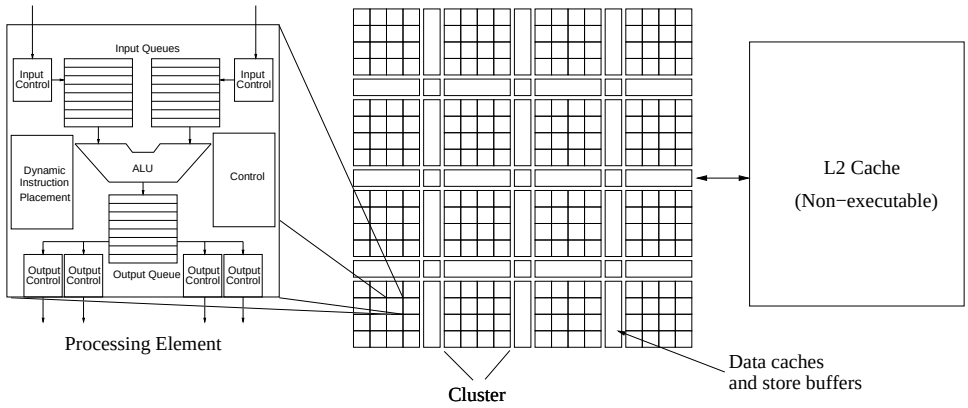


Figure 4: A simple architecture to execute fetch-less instruction sets. Note that the "processor" is nothing more than a cache for instructions. An instruction is no longer just a set of static bits, but rather a place to hold all relevant architectural state (data, control, etc) for all dynamic instances of that instruction.

factor of four, and the final instruction count overhead with folding varies from 20% to 140%. Although the overhead is substantial, the results in Section 4 demonstrate that the impact on performance is much smaller.

Part of this overhead is due to conservative assumptions necessary for the binary-translation process. We believe that overhead will decrease substantially after we complete a native WaveScalar compiler that will have more information about program structure and include aggressive optimizations to reduce overhead.

3.2 The WaveCache: a practical implementation for today

In this section, we describe a small WaveCache that can be built with current technology and can execute WaveScalar binaries. The WaveCache functions as an executable instruction cache. There is no central computing core, no issue queue, and no register file. Instead of fetching instructions from a conventional I-Cache into a processing core, the WaveCache executes instructions *in place* and in dataflow fashion. We evaluate the performance of this architecture in Section 4.

Like any architecture, the WaveCache's goal is high performance from a given area of silicon. Superscalars accomplish this by constructing complex communication infrastructure around a few functional units in an attempt to keep them continually busy. This is the correct approach when functional units are expensive and wires are cheap, but this is no longer the case (Section 2). The WaveCache, in contrast, does not focus on keeping its functional units busy. Instead, it spreads computational resources throughout the chip to reduce the complexity and latency of the communication infrastructure.

Figure 4 is a block diagram of the WaveCache. The WaveCache is a grid of approximately 2K processing tiles arranged into clusters of 16 tiles each. Each tile contains dynamic configuration logic to control instruction placement, input and output queues for instruction operands, communication logic, and a functional unit.

Each tile also contains buffering and storage for 8 different instructions, bringing the total WaveCache capacity to 16 thousand instructions – equivalent to a 64Kbyte instruction cache in a modern RISC machine. Total storage, however, is close to four megabytes when the input and output operand queues for each instruction are accounted for.

In addition to the instruction operand queues, the WaveCache contains several store buffers and traditional level 1 data caches. Each dynamic wave is bound to a nearby store buffer that processes its memory requests, and as waves complete, the store buffer of the next wave is triggered to proceed. The caches access

DRAM through a conventional unified, non-intelligent L2 cache.

Within a cluster, the 16 tiles communicate via a set of shared buses. For communication between clusters, the WaveCache uses a dynamically routed grid-based network. Tiles within the same cluster receive results at the end of the clock cycle during which they were computed. Data that must cross clusters incur a one-cycle penalty per cluster crossed. An alternative to this arrangement would be a single tile operating in isolation. Several prior tiled architectures take this approach [43, 5]. However, clustering allows the WaveCache to trade-off the advantages of long wires against their negative effect on cycle time. By varying the size of the cluster, we can tune the WaveCache's cycle time to maximize total performance.

A tile in this framework is extremely simple. It contains a basic functional unit, buffers for operands, and a network interface. We estimate that the integer execution logic and network interface will be a mere 50,000 logic transistors, with an additional 4 Kbytes of memory for input and output operand storage. This number is pessimistically calculated from examining the size of early RISC processors [44]. Within each cluster we assume that a single tile supports floating point operations and consumes an additional 500,000 transistors [45]. Finally, attached to each cluster is a small L1 data cache of 32 Kbytes. Using these figures, we estimate that no more than 250 million transistors will be used for logic, another 200 million for operand storage (4MBytes total), and another 100 million for data caches. These numbers are aggressive for current technology, but are within the scope of the forthcoming Madison chip (500 million transistors total). Even assuming that our estimates are off by a factor of two, which safely accounts for additional items such as cache control, replacement, and data cache coherence, such a device is about one billion transistors, which is similar to chips currently under development at Intel [46].

The WaveCache has an instruction memory management policy analogous to a conventional cache replacement policy. Instructions are brought into the WaveCache in precisely the same manner they are brought into a traditional cache – by way of a cache miss. When instructions already in the cache send results to instructions not currently available, a load request is sent to the cache controller. The controller locates a free node (or makes a node free by writing back an instruction and any partially computed state associated with it), and places the new instruction at that node. Instruction placement is critical to reducing inter-instruction latency, so placing dependent instructions within the same cluster is desirable.

For WaveCaches of this small size, the dependent tiles can simply be notified of the new instruction's location. Future, larger scale and more distributed systems will operate a discovery protocol, much like wide area networks.

Prefetching entire basic blocks or functions into the WaveCache could avoid loading instructions one at a time. Once in this cache, the instructions execute in place, following the WaveScalar execution algorithm (Section 3.1).

In Figure 2, the right side shows the dataflow graph mapped onto a small WaveCache.

4 Results

In this section, we explore the performance limits of the WaveScalar ISA and the WaveCache. We investigate four aspects of execution: The overhead due to WaveScalar instructions, WaveCache performance relative to a superscalar, the sensitivity of the WaveCache to cluster size, and the potential effectiveness of control and memory speculation.

4.1 Methodology

For both the WaveCache and superscalar we examine two configurations: An *ideal* configuration which assumes arbitrarily fast communication and a more *realistic* configuration with reasonable communication costs.

For our baseline WaveCache configuration, we use the system described in Section 3.2. The ideal WaveCache contains a single, infinitely large cluster, so all communication is single-cycle. The realistic WaveCache has 16 processing element per cluster, and we place instructions statically into clusters using a simple greedy strategy that attempts to place dependent instructions in the same cluster. We expect to achieve better results in the future using a dynamic placement [47] algorithm to improve layout.

Our realistic superscalar machine uses a 15 pipeline stage, 16-wide out-of-order processing core, with 1024 physical registers and a 1024 entry issue window with oldest-instruction-first scheduling. Its core uses an aggressively pipelined issue window and register file similar to what is described in [48], to reduce critical scheduling/wake-up loop delays. The core also includes a gshare branch predictor [49], store buffer, and perfect (16 ported) cache memory system. Since the pipeline is not partitioned, 15 cycles is aggressive given the size of the register file and issue window and width of the machine.

The ideal superscalar uses a shorter, seven-stage, pipeline to ignore the cost of accessing the large register file and issue queue. This decreases the critical load-use and branch misprediction loops [50]. Otherwise it is identical to the realistic machine. In all architectures (both superscalar and WaveCache) we assume a perfect L1 data cache. The superscalar is also provided with a perfect L1 instruction cache.

To perform a fair comparison to the superscalar design and leverage existing compiler infrastructure, we used a custom binary re-writing tool to convert Alpha binaries into the WaveScalar instruction set – effectively demonstrating the feasibility of binary translation from Von-Neumann to dataflow computing models.

We compiled a set of benchmarks using the Compaq cc (v5.9) compiler on Tru64 Unix, using the `-O4 -unroll 16` flags. The benchmarks are *vpr*, *twolf*, and *mcf* from SPECint2000 [51], *equake* and *art* from SPECfp2000, *adpcm*, and *mpeg2encode* from mediabench [52], and *fft*, a kernel from Numerical Recipes in C [53]. Our current simulation tool chain limits us to studying 1M instruction traces, so we carefully select and validate our starting points for each application. First, we use gprof on a native Alpha system to identify the region of code well after application startup. We collect a 1M instruction trace starting at this point. We test the validity of this trace by executing this short trace and a longer 100M instruction trace on our superscalar simulator. The difference in superscalar performance between the short and long traces is less than 5%. We then use our binary translator to converts the Alpha executable into the WaveScalar ISA, and we use a trace of the same subset of execution to drive a timing simulator of the WaveCache.

We report the results in terms of *Alpha-equivalent instructions per cycle* (AIPC). For the WaveCache measurements we carefully distinguished between instructions from the original Alpha binary, and those added by the Alpha-to-WaveScalar binary re-writer. Our binary rewriting tool introduces many new instructions (ϕ^{-1}, WAVE-COORDINATE, etc.), but these are ***not*** counted in any of the throughput measurements. Thus we use AIPC, because it fairly compares the amount of application-level work performed by each processor.

We expect WaveCache systems to achieve faster clock rates than the superscalar we are comparing against, but in this study we ignore this effect, both to be pessimistic about WaveCache performance and because we cannot yet reliably quantify the difference in clock rate.

4.2 Overhead

Translating Alpha applications into WaveScalar binaries can introduce significant overhead in terms of additional instructions. In Section 3.1.5 we discussed the increased static size of WaveScalar binaries, here we quantify their effect on performance. Added WaveScalar instructions affect performance primarily because they increase the critical path.

Table 1 shows the percentage increase in dynamic instruction count for each of our applications relative to the original Alpha binary. The mean increase is 40%, but the overhead can vary dramatically. For instance, *vpr* suffers a 76% increase in instructions executed, while *equake* and *adpcm* increase by less

	Static overhead inst. count (%)	Dynamic overhead inst. count (%)	Zero overhead speedup (%)	Wave size (inst.)
vpr	83.9	76.5	11.7	19.5
twolf	97.5	69.1	11.6	10.3
mcf	108.0	67.6	24.1	17.5
equake	17.5	0.2	0.6	81.5
art	100.1	45.6	15.7	14.8
adpcm	29.4	<0.1	0	51.2
mpeg	14.2	2.6	1.8	399.8
fft	67.7	50.9	14.6	22.9

Table 1: **WaveScalar overhead and wave size:** The first two columns are static and dynamic overhead from WaveScalar instructions. Column three is the speedup with zero-latency WAVE-COORDINATE, WAVE-RENAME, and ϕ^{-1} instructions, relative to the baseline WaveCache with a single, infinite cluster. The last column measures wave size.

than 1%. The increase for *vpr* reflects its complicated control and the corresponding large number of ϕ^{-1}, WAVE-RENAME, and WAVE-COORDINATE instructions. *equake* and *adpcm* grow very little, because they have simple control flow and benefit significantly from loop unrolling.

While the overhead may seem large, its impact on performance is less dramatic, because few of the added instructions are on the critical path; removing the latency due to overhead instructions improves performance by only 11% on average. As we mentioned in Section 3 a native WaveScalar compiler could generate code better tuned to the WaveCache.

4.3 Comparison to superscalar

Figure 5 compares the WaveCache to the superscalar. The bars are split between the ideal and realistic configurations. For the models with realistic performance, the WaveCache outperforms the superscalar by a factor of 3. For highly loop parallel applications, such as *equake*, and the applications from the mediabench programs, the WaveCache is over 4 times faster than the comparable superscalar. With ideal communication, the WaveCache is only 50% faster. The difference in performance between the ideal and realistic cases suggests that the WaveCache tolerates communication costs better than the superscalar.

In Section 3 we argued that large waves are desirable for good performance. Fully understanding the effects of wave size requires further study, but the data in table and Figure 5 demonstrate that wave size correlates with higher AIPC: the benchmarks with the largest waves (*adpcm*, *equake*, and *mpeg*) also achieve the highest AIPC.

4.4 Cluster size and instruction layout

One of the WaveCache's goals is to communicate efficiently and achieve high performance without resorting to long wires. We now examine this aspect of the WaveCache in detail.

Recall that in WaveCache configurations, processing elements are arranged in clusters and that within these clusters communication takes a single cycle. Since larger clusters require longer wires and, therefore, a slower clock, cluster size is a key parameter. Figure 6 shows performance results for clusters ranging in

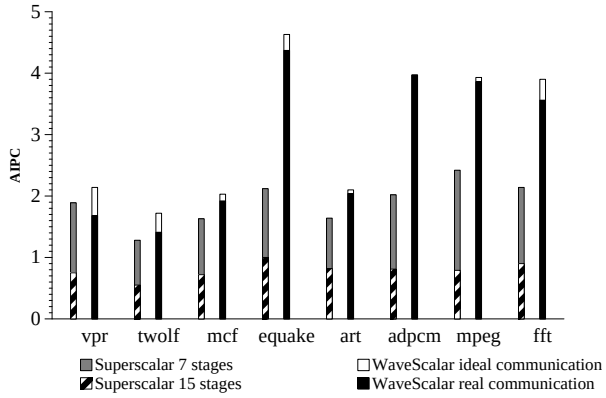


Figure 5: **Superscalar vs. WaveCache:** Each application is evaluated on the superscalar (left bar) and the WaveCache (right bar). The lower portion of the bars represents performance with realistic communication costs, while the upper portion adds in performance with ideal communication.

size from one to sixty-four processing elements. It also includes a configuration with a single, infinite cluster as an upper bound.

Overall, the performance of 16-processing-element clusters is 90% of the infinite case. This is a substantial improvement over 4-element clusters (71%), and 64 elements per cluster provides only incremental improvement (96%). Sixteen-element clusters do a fine job of capturing dataflow locality. For each benchmark, fewer than 15% of values leave their cluster of origin, and fewer than 10% must cross more than one cluster to reach their destination. Unless noted, the rest of the data we present in this section is for a 16-element cluster configuration.

Also interesting is performance with isolated processing elements. Using singleton clusters reduces performance by 54%. While this may seem like a dramatic drop, a single-element cluster WaveCache still outperforms a 15-stage superscalar by an average of 52%. Tiny clusters also reduce wire length and increase the potential clock rate. Accounting for a faster clock in our performance estimates is the subject of future study.

4.5 Control and memory speculation

Speculation is an important aspect of superscalar design, and modern processors contain speculative mechanisms. Among them, control speculation and memory independence predictions [41] provide particularly large performance gains.

Figure 7 evaluates WaveCache performance with both kinds of speculation. For each benchmark, the first bar represents baseline performance, while the others measure WaveCache performance with a variety of prediction schemes: perfect branch prediction, perfect wave prediction (described below), perfect memory disambiguation, and combinations of the three. We collected simular data for the superscalar and discuss it as needed.

4.5.1 Control speculation

Control speculation dramatically increases superscalar performance [54]. We investigate two control speculation methods for the WaveCache in the hope of achieving similar benefits. In the first, perfect branch prediction, ϕ^{-1} instructions steer values to the correct output without waiting for the selector input. The second, perfect *wave prediction*, allows the wave number value to pass directly from one WAVE-COORDINATE

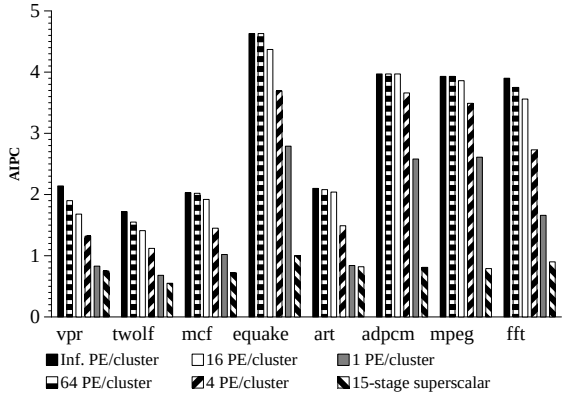


Figure 6: **Communication Costs:** For each benchmark, the first bar is the baseline ideal WaveCache performance. The next three bars measure performance with smaller clusters and, consequently, slower inter-cluster communication. For comparison, the final bar is the performance of the realistic superscalar.

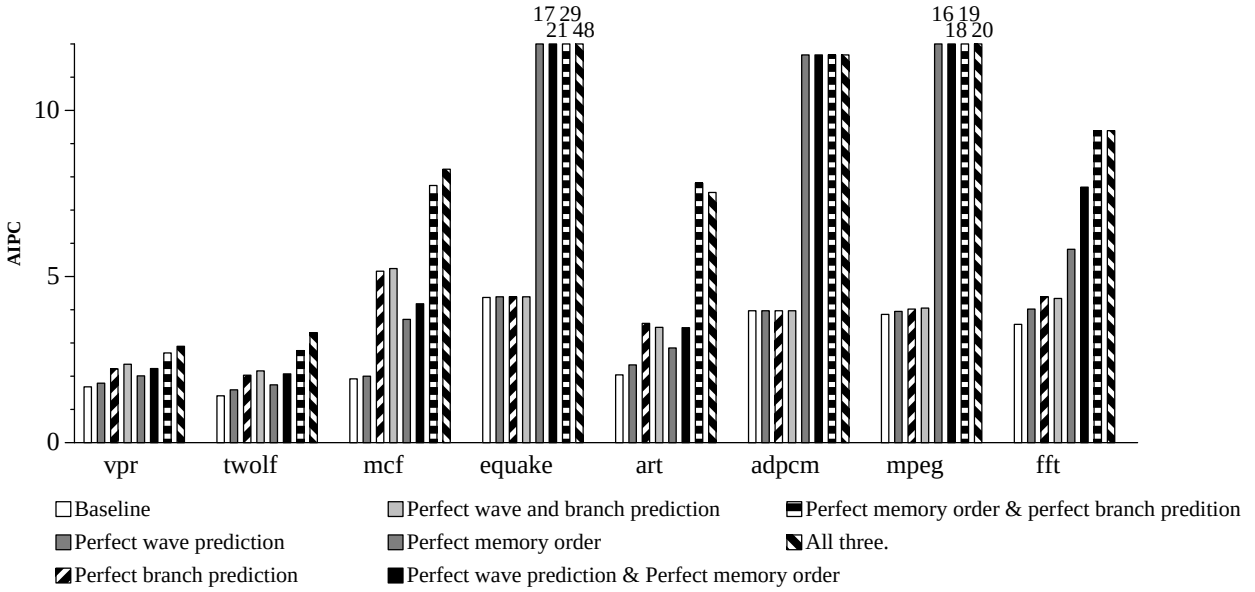


Figure 7: **WaveCache Speculation:** Comparison of the baseline realistic WaveCache configuration with a variety of speculation schemes. For *art* and *fft*, it appears that wave speculation hurts performance in some cases. This anomaly is due to limiting the number of executing waves (see Section 4.5.1).

instruction to the next instead of percolating through multiple ϕ and ϕ^{-1} instructions. The simulator cannot perform truly "perfect" wave speculation. Instead we bound the number of executing waves to 1000. Our measurements show the effect of this simplification is less than 10%.

Perfect branch prediction and wave prediction increase performance for the realistic WaveCache by 37% and 7%, respectively, and a combination of the two yields a 38% increase. This suggests that WaveCache branch prediction schemes deserve closer study.

Although the gains from wave speculation are small, the technique is interesting, since it is similar in some respects to thread speculation systems [55, 56]. This suggests, that many coarse-grained speculation schemes described elsewhere [55, 57] might fit elegantly into the WaveCache.

4.5.2 Memory speculation

Most programming languages require a memory model with completely sequentialized memory operations. In reality, however, a processor can execute memory accesses out of order, if the accesses are to different locations. To measure the potential value of this approach, we added perfect memory disambiguation to both the WaveCache and superscalar simulators.

Memory disambiguation gives a substantial boost to WaveCache performance, increasing it by an average of 61% and out-performs the superscalar with perfect disambiguation on all of the benchmarks by 170%.

Memory speculation not only provides large performance gains by itself, but also enhances branch prediction and wave prediction. Without memory disambiguation, wave prediction increases AIPC by 7%, but with disambiguation, it provides a 16% increase. Likewise, branch prediction increases AIPC by 63% in the presence of perfect disambiguation. Finally, all three forms of speculation combined, increase WaveCache performance by 170%.

Although we cannot hope to achieve these results in practice (the predictors are perfect), speculation can clearly play an important role in improving WaveCache performance. On the other hand, we have also shown that the WaveCache does not require speculation to achieve high performance.

5 Related work

WaveScalar builds upon several ground-breaking studies in both dataflow and Von-Neumann processing. In this section we place WaveScalar in the context of previous work and describe where it extends prior projects and stands in contrast to them.

5.1 Dataflow

Dataflow computing is perhaps the best studied alternative to the Von Neumann model of computation. The first dataflow architectures [6, 20] appeared in the mid to late 70's, and in the late 80's and early 90's there was a notable revival [21, 22, 23, 24, 25, 26]. The dataflow work of the late 80's and early 90's made it clear that high performance dataflow machines were difficult to build. Culler et. al. [58] articulated this difficulty as a cost/benefit problem and argued that dataflow suffers from two fundamental problems, both of which have to with the top of the memory hierarchy.

First, the memory hierarchy limits the amount of latency a dataflow machine can hide. A processor can only hide latency (and keep busy) by executing instructions whose inputs are at the top level of the memory hierarchy. If the top of the memory hierarchy is too small, the processor will sit idle, waiting for the inputs to arrive.

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

Culler's arguments were sound at the time, but they are bound to the assumption that execution resources are expensive and that hiding latency and keeping the processors busy are the keys to performance. Given that commercial microprocessors ship with less than 10% of the die area devoted to execution, this persecutive is no longer up to date. The key efficiency point is total performance per chip, independent whether this performance is achieved through hundreds of execution cores (WaveCaches), or massive communication networks (superscalars). We have demonstrated that the WaveCache can effectively exploit locality in communication patterns to increase performance for a given die area.

This does not mean, however, that the memory hierarchy is unimportant, but in recent years the size of on-chip memories has soared. The PA-RISC 8700 has 2.25MB of L1 caches [59] and the forthcoming Madison chip from Intel will include a 6MB L2 cache [60]. Since each WaveCache processing element contains a small memory, the WaveCache can simultaneously utilize an enormous amount of on-chip storage with very short access times.

Traditional dataflow work has too often relied upon alternative memory models [61, 62], which, while enhancing performance, limit acceptance. Wave-ordered memory make WaveScalar a uniquely different sort of dataflow instruction set: Through an aggressive dynamic wave and static-ordering process WaveScalar supports traditional memory semantics, including side-effects, indirection and aliasing. In our opinion, supporting traditional programming languages is required for instruction sets and new architectures to be successful.

The WaveScalar ISA builds upon some of the original program representations used in [19, 27], which was derived from earlier compiler and theory work [33]. The intermediate compiler language, Pegasus [63], for NanoFabrics [64] adapts these ideas as well. The Pegasus researchers transform an entire application into a static dataflow graph and map it onto a large spatial fabric of molecular electronics that can operate like an FPGA. To date, the system is limited to a static dataflow model, although the early discussions of moving to a partially dynamic system are in [63].

Treating the entire device as an FPGA and mapping the entire application to it is intriguing, but our cache-based approach has the advantage of being able to localize in physical space the dependencies of an application at run time, instead of compile time. This is particularly useful when co-locating computation and data. By keeping instructions and data at a fixed location in the NanoFrabric, memory operations sometimes have to travel quite long distances, repetitively. It is not clear yet how the NanoFabric work intends to really handle memory ordering; however, the wave-ordered memory would work for their system.

5.2 TRIPS

The TRIPS / GPA [3] processor and WaveScalar are attacking the same technology challenges, and tend to use the same terminology to describe aspects of their designs. However, the only architectural feature TRIPS and WaveScalar share is the use of direct links between instructions of the same hyper-block (or wave). TRIPS is an innovative way to build a Very Long Instruction Word (VLIW) processor from next generation silicon technology. A VLIW bundles instructions horizontally to be executed in parallel. The TRIPS processor makes the keen insight that between subsequent VLIW instructions is a significant amount of dependence. Hence, it bundles groups of VLIW instructions together vertically and describes their dependencies explicitly instead of implicitly through registers. Next it statically schedules them onto a physically horizontal and vertical VLIW-like set of functional units. For the most part, a traditional centralized register file is used to pass data items between hyperblocks; however, work is ongoing within the TRIPS project to develop methods to stitch data dependences dynamically between adjacent hyper-blocks [65]. It will be interesting to see how far this notion can be pushed by the TRIPS project without transforming all the way

to a dataflow model of computation like WaveScalar.

WaveScalar offers three key advantages over the existing TRIPS architecture. First, it makes a clean break away from a single program counter. This exposes more parallelism through the execution model, without relying upon trace scheduling compilers. Lam et al [15] illustrate that being forced to process control dependencies in program order severally constrains ILP. Unlike TRIPS, WaveScalar, breaks away from this constraint, taking a step towards what Lam terms the "CD-MF" (control dependence with multiple flows) model of computation. Yet it does this *without* speculation. It achieves it by virtue of being based on a dataflow model of computation and wave-memory ordering. The second advantage WaveScalar offers over TRIPS is that instructions are designed to execute in-place in an intelligent memory system. This allows for the construction of a cache to exploit dataflow locality across far-flung sections of an application, not just locally adjacent hyperblocks. Furthermore the dynamic nature of the WaveCache allows it to optimize these instruction placements locally with runtime information. TRIPS relies upon static scheduling of dependent instructions within a single hyperblock. Finally, the WaveScalar memory model exposes memory parallelism across hyperblocks (i.e. with in wave). Like all Von-Neumann machines, memory ordering occurs through a program counter with TRIPS, relying upon good control prediction for performance. Only time and more experiments will tell which model of computation (dataflow or Von Neumann) and which architecture (WaveScalar or TRIPS) is the right solution.

5.3 RAW and SmartMemories

The idea of computing with ten's to hundred's of nodes on a chip is not new. The RAW [43] and SmartMemories [5] project use a tiled node architecture. While pictorially WaveScalar systems might appear like yet another tiled architecture, there are several key differences. On some level, both RAW and SmartMemories are really chip-multiprocessors, except that they have sophisticated and novel communication facilities tied into their processing cores and memory systems. Both architectures use a processor connected to a memory for each node.

The RAW project puts forth two programming models. The first, similar to SmartMemories, is that of an advanced chip multiprocessor; the second is that of a speculative threaded machine [66]. Unfortunately, the inter-node communication latency between tiles in RAW is extremely high, compared to classic inter-functional unit latencies in superscalars (but compared to a conventional DSM it is quite low).

6 Future work

Since our investigation of the WaveScalar ISA and a possible WaveCache implementations is just beginning, there are far more questions than answers. In this paper we presented our early results and initial investigations. Below, we outline some of the areas of future work.

Microarchitecture: Chief among the unexplored microarchitectural issues are data caching and deadlock avoidance. Our current architecture distributes the datacache into several small caches throughout the architecture. Our plan is to apply an existing directory-based cache coherence protocol [67, 68] to these on-chip cache. We are building a model of this protocol and cache hierarchy into our simulation framework, to explore its effect on performance. Additionally, the current WaveCache architecture uses a dynamically routed switched network, similar to [5]. Without the ability to drop packets within the network, these networks can deadlock. Our current work uses a reliable acknowledge/resend mechanism between nodes, but in the future we will investigate integrated checkpointing mechanisms. Checkpointing may have additional uses besides deadlock avoidance in the area of coarse grained speculation.

Compiler: A top priority is to replace the Alpha ISA to WaveScalar ISA binary rewriter with a back-end for an existing C/C++ compiler. Using a native compiler will generate more efficient code and allow

us to explore opportunities and problems unique to WaveScalar, such as compiler-directed wave number management and direct execution of single static assignment [38] code using ϕ instructions.

Speculation: In Section 4 we demonstrated that WaveScalar can see significant benefit from several kinds of speculation. WaveScalar systems can speculate by initiating speculative execution at a node with one message and completing (or squashing) it with a second message once the correct path or value is available. We are currently exploring several existing techniques for control and fine and coarse grained memory speculation [55, 40, 69] and determining whether and how to integrate them into our design.

Defect tolerance: Large WaveCache systems will suffer from defective nodes, clusters, and communication networks. The fact that our architecture is uniform and decentralized means that we should be able to map around such defective nodes with little performance loss. Exploring the effects of defective nodes and dynamic faults will be a long-term focus of WaveScalar research.

Threads: The WaveScalar instruction set and WaveCache architecture are ideal tools to build threaded machines. Currently, there are two classes of threaded processors, simultaneous multithreading machines and chip multiprocessors. The fundamental difference between the two is whether resources are partitioned statically between the threads or shared dynamically. In the WaveCache this is simply a parameter to the WaveCache replacement policy: for the SMT [70] model, all the threads compete for the available processing elements; in the chip multiprocessor [71] model the WaveCache confines each thread to a portion of the grid. By simply adding a THREAD-ID to each wave number, and modifying the memory interface accordingly, a vast array of multithreading strategies become possible.

7 Conclusion

In this paper we have presented WaveScalar, a new dataflow execution instruction set with several attractive properties. In contrast to prior dataflow work, WaveScalar provides a novel memory ordering model, wave-ordered memory, that efficiently supports mainstream programming languages on a true dataflow computing platform without sacrificing parallelism. Dividing the program into waves, combined with the WAVE-RENAME and WAVE-COORDINATE instructions provide decentralized, inexpensive, software-controlled tag management. In practice, WaveScalar programs run in a distributed computation substrate called the WaveCache that co-locates computation and data values to reduce communication costs and exploit dataflow locality.

The performance of our initial WaveCache is promising. The WaveCache's ability to exploit parallelism usually hidden by the Von Neumann model, leads to a factor of 2-4 performance increase in our limit study on the SPEC and mediabench applications when compared to an aggressively configured superscalar processor. It does this, without speculation, in a communication-scalable architecture that provides several opportunities for further study and refinement.

References

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

[2] T. M. Austin, "DIVA: A reliable substrate for deep submicron microarchitecture design," in *32nd International Symposium on Microarchitecture*, 1999.

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

[4] 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.

[5] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz, "Smart memories: A modular reconfigurable architecture," in *International Symposium on Computer Architecture*, 2002.

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

[7] W. Hunt, "Introduction: Special issue on microprocessor verification," in *Formal Methods in System Design*, Kluwer Academic Publishers, 2002.

[8] D. Weiss, J. J. Wuu, and V. Chin, "The on-chip 3mb subarray based 3rd level cache on an itanium microprocessor," in *IEEE International Solid-State Circuits Conference, 2002*, 2002.

[9] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," in *IBM Journal*, 1967.

[10] R. M. Keller, "Look-ahead processors," *ACM Computing Surveys (CSUR)*, vol. 7, no. 4, pp. 177–195, 1975.

[11] H.-S. Kim and J. E. Smith, "An instruction set and microarchitecture for instruction level distributed processors," in *29th International Symposium on Computer Architecture*, 2002.

[12] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th international symposium on Computer architecture*, ACM Press, 1997.

[13] "Map-ca datasheet," June 2001. Equator Technologies.

[14] H. Sharangpani, "Intel Itanium processor core," in *Hot-Chips*, 2000.

[15] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in *19th International Symposium on Computer Architecture*, 1992.

[16] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, (New York, NY), pp. 176–189, ACM Press, Apr. 1991.

[17] A. M. Kumar, "Measuring parallelism in computation-intensive scientific/engineering applications," *IEEE Transactions on Computers*, vol. 37, no. 9, pp. 1088–1098, 1988.

[18] A. A. Nicolau and J. Fisher, "Using an oracle to measure potential parallelism in single instruction stream programs," in *Proceedings of 14th Annual Microprogramming Workshop*, pp. 171–182, 1981.

[19] J. B. Dennis, "Dataflow supercomputers," in *IEEE Computer*, IEEE, November 1980.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] K. J. Ottenstein, R. A. Ballance, and A. B. Maccabe, "The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages," in *Proceedings of the conference on Programming language design and implementation*, pp. 257–271, ACM Press, 1990.

[29] R. Nikhil, "The parallel programming language id and its compilation for parallel machines," in *Proceedings of the Workshop on Mazzive Paralleism: Hardware, Programming and Applications*, Acamedic Press, 1990.

[30] 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.

[31] J. T. Feo, P. J. Miller, and S. K. Skedzielewski, "Sisal90," in *Proceedings High Performance Functional Computing*, April 1995.

[32] S. Murer and R. Marti, "The fool programming language: Integrating single-assignment and object-oriented paradigms," in *Proceedings of the European Workshop on Parallel Computing*, IOS Press, 1992.

[33] J. B. Dennis, "First version data flow procedure language," Tech. Rep. MAC TM61, MIT Laboratory for Computer Science, May 1991.

[34] E. A. Ashcroft and W. W. Wadge, "Lucid, a nonprocedural language with iteration," *Communications of the ACM*, vol. 20, no. 7, pp. 519–526, 1977.

[35] J. R. McGraw, "The val language: Description and analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 1, pp. 44–82, 1982.

[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] 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.

[39] 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.

[40] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi, "Speculative versioning cache," in *Symposium on High Performance Computer Architecture*, pp. 195–205, 1998.

[41] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, "Dynamic speculation and synchronization of data dependences," in *International Symposium on Computer Architecture*, pp. 181–193, 1997.

[42] M. Franklin and G. S. Sohi, "ARB: a hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, vol. 45, pp. 552–571, May 1996.

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

[44] D. A. Patterson and C. H. Sequin, "RISC i: A reduced instruction set vlsi computer," *Proceedings of the 8th Annual International Symposium on Computer Architecture*, May 1981.

[45] R. Rogenmoser, L. O'Donnell, and S. Nishimoto, "A dual-issue floating-point coprocessor with simd architecture and fast 3d functions," in *Proceedings of ISSCC 2002*, IEEE, 2002.

[46] P. Otellini, "Driving convergence through silicon integration," Fall 2002. Intel Developer Forum.

[47] S. Swanson, K. Michelson, and M. Oskin, "Configuration by combustion: Online simulated annealing for dynamic hardware configuration," in *ASPLOS X Wild and Crazy Idea Session*, Oct. 2002 2002.

[48] M. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, and S. W. K. P. Shivakumar, "The optimal useful logic depth per pipeline stage is 6-8 fo4," in *Proceedings of the 29nd annual international symposium on Computer architecture*, ACM Press, 2002.

[49] S. McFarling, "Combining Branch Predictors," Tech. Rep. TN-36, Digital Equipment Corperation, June 1993.

[50] M. S. Hrishikesh, D. Burger, N. P. Jouppi, S. W. Keckler, K. I. Farkas, and P. Shivakumar, "The optimal logic depth per pipeline stage is 6 to 8 fo4 inverter delays," in *Proceedings of the 29th annual international symposium on Computer architecture*, pp. 14–24, IEEE Computer Society, 2002.

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

[52] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *International Symposium on Microarchitecture*, pp. 330–335, 1997.

[53] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[54] S. Swanson, L. McDowell, M. Swift, S. Eggers, and H. Levy, "An evaluation of speculative instruction execution on simultaneous multithreaded processors," *Submitted for publication*, 2002.

[55] 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.

[56] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, and B. Towles, "IMAGINE: Signal and image processing using streams," in *Hot Chips 12: Stanford University, Stanford, California, August 13–15, 2000* (IEEE, ed.), IEEE Computer Society Press, 2000.

[57] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Willey, M. Chen, M. Kozyrczak, and K. Olukotun, "The Stanford Hydra CMP," in *Hot Chips 11: Stanford University, Stanford, California, August 15–17, 1999* (IEEE, ed.), IEEE Computer Society Press, 1999.

[58] D. E. Culler, K. E. Schauser, and T. Eicken von, "Two fundamental limits on dataflow multiprocessing," in *Proceedings of the IFIP Working Group (Concurrent Systems) Conference on Architectures and Compilation techniques for fine and medium grain parallelism*, Elsevier, Jan 1993.

[59] HP, "Pa-risc 8x00 family of microprocessors with focus on the pa-8700," 2000. Hewlitt Packard.

[60] H. Cornelius, "High-performance computing on intel architecture," 2002. Intel Corp.

[61] Arvind, R. Nikhil, and K. K. Pingali, "I-structures: Data structures for parallel computing," *ACM Transaction on Programming Languages and Systems*, vol. 11, no. 4, pp. 598–632, 1989.

[62] 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.

[63] M. Budiu and S. C. Godstein, "Pegasus: An efficient intermediate representation." CMU Technical Report: CMU-CS-02-107, 2002.

[64] S. C. Goldstein and M. Budiu, "Nanofabrics: Spatial computing using molecular electronics," in *International Symposium on Computer Architecture*, 2001.

[65] "Personal communication with doug burger and steve keckler," 2002.

[66] A. Agarwal, S. Amarasinghe, R. Barua, M. Frank, W. Lee, V. Sarkar, D. Srikrishna, and M. Taylor, "The raw compiler project," in *Proceedings of the Second SUIF Compiler Workshop*, August 1997.

[67] C. K. Tang, "Cache design in the tightly coupled multiprocessor system," in *In AFIPS Conference Proceedings, National Computer Conference*, 1976.

[68] J. Archibald and J. L. Baer, "An economical solution to the cache coherence problem," in *The 11th Annual International Symposium on Computer Architecture*, pp. 355–362, 1984.

[69] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Proceedings of the 29th annual international symposium on Computer architecture*, pp. 123–134, IEEE Computer Society, 2002.

[70] D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous multithreading: Maximizing on-chip parallelism,” in *Proceedings of The International Symposium On Computer Architecture*, 1995.

[71] 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 (ASPLOS IIV)*, 1996.