Dataflow: The Road Less Complex

Steven Swanson Ken Michelson Andrew Schwerin Mark Oskin

University of Washington
Computer Science and Engineering
{swanson,ken,schwerin,oskin}@cs.washington.edu

Abstract

Silicon technology will continue to provide an exponential increase in the availability of raw transistors. Effectively translating this resource into application performance, however, is an open challenge. 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 scalable, low-complexity/high-performance processors. WaveScalar is unique among dataflow architectures in efficiently providing traditional memory semantics. At last, a dataflow machine can run "real-world" programs written in any language.

This paper introduces the WaveScalar instruction set and an implementation based on current technology that is much simpler than current out-of-order designs. We evaluate our design's performance potential using the SPEC and mediabench applications. Our results demonstrate a 30-130% performance improvement compared to an aggressively configured out-of-order 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 into 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.

The complexity of these 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 [2] and verification costs are increasing. The omnipresent requirement that these new circuits operate at very high frequencies only exacerbates these problems.

Like the memory wall, the processor scaling wall has motivated a number of research efforts [3, 4, 5, 6]. These efforts all augment the existing von Neumann model of computation by providing redundant checking mechanisms [3], by exploiting compiler technology for limited dataflow-like execution [4], or by efficiently exploiting coarse-grained parallelism [6, 5]. In this paper we propose another approach, WaveScalar, that does not rely upon the von Neumann model.

At its core, WaveScalar is a dataflow instruction set and computing model [7]. The dataflow execution model is fundamentally decentralized, because each instruction decides when it should fire by examining its available inputs. As a result, dataflow execution can utilize a vast amount of parallelism if it is present in the program. By contrast modern von Neumann-style processors hide much of the available parallelism by requiring the program counter to generate a linear sequence of instructions to feed into an out-of-order core.

The key difference between WaveScalar and prior dataflow architectures is that WaveScalar efficiently supports traditional von Neumann-like memory semantics in a dataflow model. Previously dataflow machines provided their own style of memory semantics and their own special dataflow languages that disallowed side effects, mutable data structures, and many other useful programming constructs [8, 9, 10]. Indeed, a memory ordering scheme that allows a dataflow machine to efficiently execute code written in general purpose, imperative languages (such as C, C++, or Java), has eluded researchers for several decades. In Section 2 we present a memory ordering scheme that efficiently enables true dataflow execution of programs written in *any* language.

The new memory ordering scheme removes the need for instruction fetch, the first of two serialization points in a superscalar processor. The second serialization point, the memory interface, is essential to ensure that memory operations commit (or appear to commit) in order, but serialized fetch serves little purpose aside from generating the order that the memory system must enforce.

Unfortunately, fetch also defines a total ordering on all instructions instead of just memory operations creating a false control dependency between each instruction and the next. Superscalar processors use large and enormously complex hardware structures to recover just a fraction of the parallelism that fetch destroys. WaveScalar provides an ordering only on memory operations, obviating the need for instruction fetch, unleashing large amounts of parallelism, and eliminating the need for the costly, centralized hardware structures that increase design time and limit clock speeds in out-of-order processors.

Other recent attempts to build scalable processors such as GPA [4] and Raw [11] have extended the von Neumann paradigm in novel ways, but they still rely on two serialization points, unnecessarily limiting the amount of parallelism they can reveal. WaveScalar abandons the program counter completely and provides serialization only where the program requires it, at memory.

In the next section we describe the WaveScalar instruction set and a simple WaveScalar processor design. Section 3 presents an initial evaluation of our WaveScalar design, and Section 4 discusses related work in this area. Finally in Sections 5 and 6, we discuss future work and conclude.

2 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.

Dataflow has a long history. The first designs appeared in the early 70's [7, 8, 12], and there was a significant revival in the 80's and early 90's [13, 14, 15, 16, 17, 18, 9]. Dataflow machines execute programs according to the dataflow firing rule, 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 provide references for some of these in Section 4.

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 in practice. Instruction communicate with one another directly by passing messages instead of writing and reading values from a central register file.

In the next two sections, we describe the WaveScalar ISA and memory ordering scheme and a sample processor design that could be built today.

2.1 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 and contemporary dataflow assembly languages[12, 8, 19, 9, 20].

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, WaveScalar targets programs written in mainstream imperative languages (such as C), instead of those written in specialized dataflow languages [21, 22, 23, 24, 10, 25, 26]. There have been some prior attempts at this [27, 28], but none adequately addressed the most difficult challenges including pointers, aliasing, and mutable data structures.

Below we describe they key aspects of the WaveScalar ISA and the its memory ordering scheme. For a more thorough description of the WaveScalar ISA and the features it shares with most dataflow instruction sets see [29].

2.1.1 Control flow

Dataflow instruction sets must make control dependencies explicit because there is no program counter. WaveScalar accomplishes this using ϕ and ϕ^{-1} instructions. Other dataflow ISAs contain essentially equivalent structures.

ϕ instructions [30] 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.

ϕ^{-1} instructions [9] do the opposite and 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. They are required for implementing loops.

These two types of instructions effectively replace branch instructions and the register file of a superscalar with dynamically routed communication networks that are under compiler control.

2.1.2 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.

A traditional dataflow machine uses *tags* to identify different dynamic instances. In WaveScalar, we aggregate tag management across a directed acyclic graph 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.

- **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 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.

The effect of these instructions is similar to register renaming in a superscalar core. They both allow many instances of the same instruction to execute simultaneously, revealing more parallelism.

2.1.3 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.

WaveScalar brings load-store ordering to dataflow computing using *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. These instructions participate in memory ordering but have no effect. In practice, MEMORY-NOP's are rare (less than 3% of instructions).

Figure 1 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 can wait 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 1 depicts the operation of a wave-ordered store buffer. Alternatively, a speculative memory system [31, 32, 33] could use the ordering to detect misspeculations.

2.2 A practical implementation for today

In this section, we outline the design of a small WaveScalar processor that can be built with current technology and execute WaveScalar binaries. We point out four problems, queue overflow, finding instructions in the processor, instruction replacement, and the design of the on-chip network, that arise in implementing WaveScalar CPUs and describe "proof-of-concept" solutions.

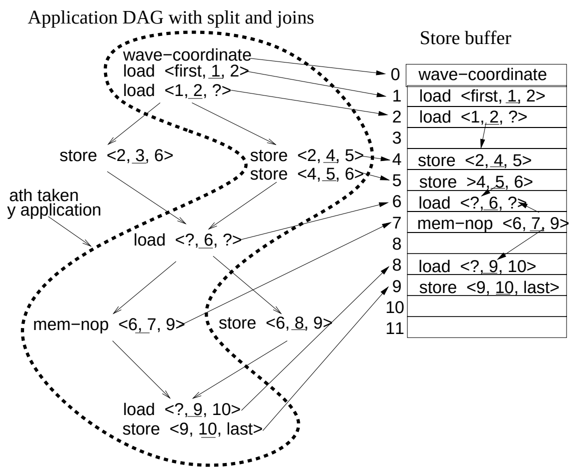

Figure 1: **Wave-ordered Memory:** Memory operations are ordered through a combination of static sequence numbers and dynamic wave numbers. The key is that on any path taken through a wave, complete knowledge about the load store order is available.

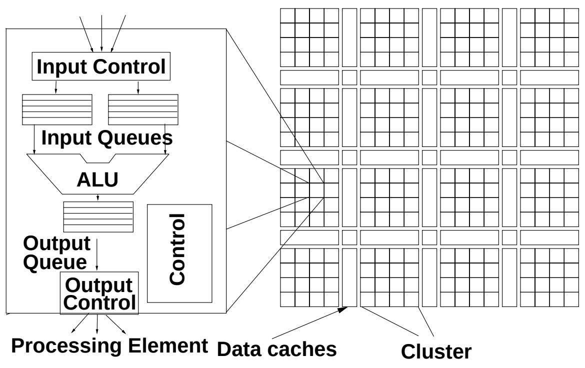

Figure 2: **A Simple WaveScalar Processor:** A simple architecture to execute the WaveScalar ISA. The processor combines clusters of processing elements (left) with small data caches and store buffers to form a computing substrate (right).

2.2.1 The basic design

Figure 2 is a block diagram of a WaveScalar core. It 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 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 processor 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, WaveScalar 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.

To execute, each instruction is bound to one of the processing elements, where it remains for possibly millions of executions. Good instruction placement is paramount for optimal performance, because communication latency depends on instruction placement.

2.2.2 Queue overflow

The WaveScalar model assumes that each instruction can buffer an unlimited number of input values and perform wave number matching among them. In reality, there is limited storage available at each processing element. Our initial solution to this problem is to store overflow values in main memory. Each input queue stores its overflow values in a portion of the virtual address space dedicated to it. The address of an overflow value is the instruction address, queue number, and wave number concatenated together. When a new overflow value is stored to memory, the processing element checks for the corresponding inputs in the other input queues (also possibly stored in main memory). If a match is found the instruction can fire. We expect in-memory matching to be rare. The precise details of overflow management are a subject of future work.

2.2.3 Finding instructions

Once an instruction is in the grid it must determine the physical location of its dependent so it can send values to them. To solve this problem the processor maintains a table in memory that maps each instruction's address in memory to a location in the grid. When an instruction arrives in the grid, it looks up the location of each of its dependents and sends the values it produces to those locations. The key is that it is acceptable for this translation process to be relatively slow, because it only needs to occur when the instruction arrives in the cache. The cost can be amortized over many executions.

2.2.4 Cache replacement

Inevitably, not all the instructions in a program will fit into the processors. We detect cache misses by defining a special "not present" location. In the location table described above, any instructions not currently in the processor has this location. When a message arrives at this special location, the destination instruction will be assigned to a processing element to receive it, and the sender will receive a message informing it that the destination's location has changed.

If there is no room for the new instruction, an instruction must be removed. Removing an instruction involves swapping out all of its state and notifying the instructions that send it values that it has left the grid. To reduce the cost of removal, instructions can volunteer to leave when their queues are empty and they have not been executed for some time. Proactively removing instructions from the grid means there will almost always be an open space in the grid for an incoming instruction.

2.2.5 On-chip communication

The on-chip network connecting the processing element clusters is an important aspect of a WaveScalar processor. We propose a simple semi-random dynamic routing scheme that attempts to route messages along the shortest possible path but occasionally perturbs messages' paths to avoid contention. In addition, the network provides an ack-based backpressure mechanism that allows instructions to keep its input queues short and avoid the "parallelism explosion" problem that can cause problems with dataflow machines [34]. Our preliminary results show that surprisingly few messages travel across the network because fewer than 11% of messages leave their cluster of origin. On average, our data suggest that fewer than 10-20 messages per thread would be in-flight on the inter-cluster network each cycle. As a result, we expect contention on the network to be minimal.

3 Results

In this section, we compare our WaveScalar processor to an aggresive superscalar design.

For our WaveScalar configuration, we use the system described in Section 2.2, except we do not model contention in the network or bound the size of the instruction input queues. It 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.

Our 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 [35], to reduce critical scheduling/wake-up loop delays. The core also includes a gshare branch predictor [36], store buffer, and 16 ported 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. Both machines have perfect memory systems.

We compile our benchmarks using the Compaq cc (v5.9) compiler on Tru64 Unix, using the `-O4 -unroll 16` flags. The benchmarks are *vpr*,

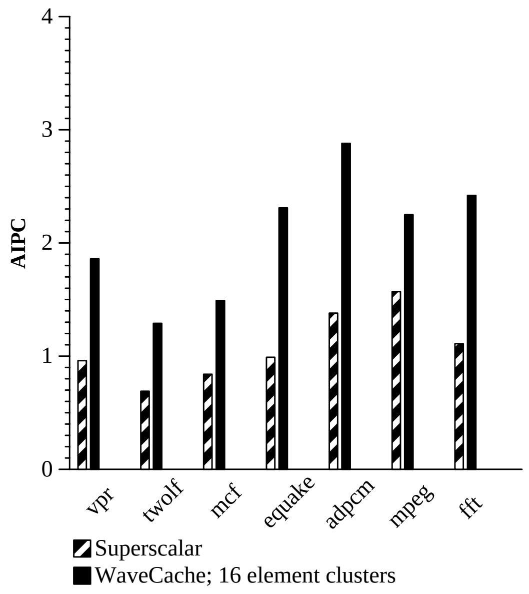

Figure 3: **Superscalar vs. WaveScalar:** Each application is evaluated on the superscalar (left bar) and the WaveScalar processor (right bar). The vertical axis is performance measured by AIPC.

twolf, and *mcf* from SPECint2000 [37], *equake* from SPECfp2000, *adpcm*, and *mpeg2encode* from mediabench [38], and *fft*, a kernel from Numerical Recipes in C [39]. We use a custom binary rewriting tool to translate Alpha binaries into WaveScalar binaries.

We report the results in terms of *Alpha-equivalent instructions per cycle* (AIPC). For the WaveScalar measurements we carefully track which WaveScalar instructions correspond to particular instructions in the alpha executable and which are added by the binary translation process. This allows us to measure the amount of work in terms of alpha instructions per cycle, a metric that fairly compares the amount of application-level work each processor performs.

Figure 3 compares the WaveScalar processor to the superscalar. The WaveScalar processor outperforms the superscalar by a factor of 1.8, on average. For highly loop parallel applications, such as mpeg, the WaveScalar processor is over 2.3 times faster than the comparable superscalar. In addition, we expect the WaveScalar processor to have faster cycle time, improving its performance further.

4 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.

4.1 Dataflow

Dataflow computing is perhaps the best studied alternative to the von Neumann model of computation. The first dataflow architectures [7, 12] appeared in the mid to late 70's, and in the late 80's and early 90's there was a notable revival [13, 14, 15, 16, 17, 18]. 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. [40] 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 perspective is no longer up to date. The key measure of efficiency is total performance per chip, independent whether this performance is achieved through hundreds of execution cores (WaveScalar), or massive communication networks (superscalars). We have demonstrated that a WaveScalar processor 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 [41] and the forthcoming Madison chip from Intel will include a 6MB L2 cache [42]. Since each WaveScalar processing element contains a small memory, the processor can simultaneously utilize an enormous amount of on-chip storage with very short access times.

The WaveScalar ISA builds upon some of the original program representations used in [8, 9], which was derived from earlier compiler and theory work [10]. The intermediate compiler language, Pegasus [43], for NanoFabrics [20] 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 [43].

4.2 TRIPS

The TRIPS / GPA [4] 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 (in TRIPS) directed acyclic graph (in WaveScalar). 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 hyperblocks [44]. 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 [45] 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 WaveScalar allows it to optimize these instruction placements locally with runtime information. TRIPS relies upon static placment of dependent instructions within a single hyperblock. Finally, the WaveScalar memory model exposes memory parallelism across hyperblocks (i.e. within wave). Like all von Neumann machines, memory ordering occurs through a program counter with TRIPS, relying upon good control prediction for performance.

4.3 RAW and SmartMemories

The idea of computing with ten’s to hundred’s of nodes on a chip is not new. The RAW [5] and SmartMemories [6] 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 [46]. 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).

5 Future work

In this paper we presented early results of our initial investigation. Our research of WaveScalar is just beginning, however, so at this point there are far more questions than answers. There are many avenues of future WaveScalar research open to investigation. We outline a broad research agenda below.

5.1 Microarchitecture and hardware

Detailed Evaluation: Our first priority is to perform a more thorough evaluation of WaveScalar processor performance both relative to superscalar designs and with respect to the key architectural parameters (e.g. cluster size, number of clusters, etc.).

After that, the most pressing unexplored microarchitectural issues are data caching and communication infrastructure. 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 [47, 48] to these on-chip caches. We are building a model of this protocol and cache hierarchy into our simulation framework, to explore its effect on performance.

The current WaveScalar architecture uses a dynamically routed switched network, similar to [6]. Without the ability to drop packets within the network, these networks can deadlock. Initially, we will use use 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.

Hardware implementation: We plan to implement a WaveScalar processor using the Bathysphere [49] emulation system. A working prototype will both demonstrate the simplicity of WaveScalar implementations and allow us to explore operating system issues, long-term performance, and fault tolerance on a real, running system.

Speculation: Preliminary results suggest 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 [50, 31, 51] and determining whether and how to integrate them into our design.

WaveScalar cache hierarchies: WaveScalar processors cache instructions for execution and, therefore, techniques that improve traditional cache performance should also improve WaveScalar performance. For example, a Level-2 WaveScalar cache might contain a vast number of instructions but matching might be relatively slow. By extension, we could construct a multi-level WaveScalar cache hierarchy. Victim buffers and instruction prefetching are also intriguing optimizations.

Online optimization: We have observed that allowing the location of instructions in the grid to evolve with program behavior can significantly improve performance. In [52] we described a approach to real-time adaptation based on simulated annealing that improves performance by between 30 and 50%, but this is only one of many possible approaches.

5.2 Instruction set and software support

Compiler: A top priority is to replace the Alpha ISA to WaveScalar ISA binary rewriter with a backend 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 [30] code using ϕ instructions.

Once the basic infrastructure is in place we will be able to explore many interesting aspects of the WaveScalar model. The WaveScalar ISA makes explicit many aspects of execution that are implicit in the von Neumann model. Memory order, control flow, instruction-to-instruction communication, and synchronization (in the form of wave number matching) are all first class entities in the WaveScalar ISA. Programmers and compilers can use the primitives to implement a huge variety of control- and data-parallel programing paradigms (vector, stream-based, micro-threads, etc.). Eventually, we hope to tune and extend WaveScalar to be "complete," that is, capable expressing any form of parallelism in a single instruction set.

Program-defined hardware interfaces: Because WaveScalar processors are naturally multithreaded, For instance, a purpose-built memory thread could could "wrap" the normal memory interface hardware and handle memory requests in application-specific ways. Applications include scatter/gather, streaming, and encrypted memory interfaces. For applications that require the normal interface, processing elements that implement these threads can be available for normal instructions.

Program-defined hardware interfaces: WaveScalar processors are naturally multithreaded. By simply adding a THREAD-ID to each wave number, and modifying the memory interface accordingly, a vast array of multithreading strategies become possible. For instance, it is possible for the programmer or the compiler to design custom threads to handle or augment tasks traditionally implemented in hardware. Applications include scatter/gather, streaming, and encrypted memory interfaces. For applications that require the normal interface, processing elements that implement these threads can be available for normal instructions.

5.3 System abstractions and parallelism

For WaveScalar CMP vs. SMT is moot: The WaveScalar instruction set and WaveScalar architecture are ideal tools for building threaded machines, because dataflow models are naturally multithreaded. 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 WaveScalar processors this is simply a parameter to the instruction replacement policy: for the SMT [53] model, all the threads compete for the available processing elements; in the chip multiprocessor [54] model WaveScalar confines each thread to a portion of the grid.

Defect tolerance: Large WaveScalar systems will suffer from defective nodes, clusters, and communication networks. The fact that our architecture is uniform and decentralized means that it 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.

Computer systems as substrate: Once we adopt a uniform substrate for computation, the distinction between the processors and the rest of the computer system's components begins to blur. For instance, it is easy to imagine building ever-larger WaveScalar processors by assembling multiple WaveScalar chips, much as we increase memory capacity by inserting more DRAM chips. Programs could spread across multiple chips, even multiple systems, and different chips could provide different functions. A WaveScalar graphics processor would have support for 3D rendering as would a WaveScalar I/O controller. To the program, all these computational resources would appear simply as different types of functional units embedded in the computational substrate. Taken to the extreme, this model of system construction unifies the namespace of what is inside the processor (execution resources), what is on the system board and beyond.

6 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 that efficiently supports mainstream programming languages on a true dataflow computing platform without sacrificing parallelism.

WaveScalar's ability to exploit parallelism usually hidden by the von Neumann model, leads to a factor of 2 performance increase on the SPEC and mediabench applications when compared to an aggressively configured superscalar processor. It does this in a communication-scalable architecture that provides many 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] W. Hunt, “Introduction: Special issue on microprocessor verification,” in *Formal Methods in System Design*, Kluwer Academic Publishers, 2002.

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

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] tech. rep.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] D. E. Culler and Arvind, “Resource requirements of dataflow programs,” in *Proceedings of the 15th Annual International Symposium on Computer architecture*, pp. 141–150, IEEE Computer Society Press, 1988.

[35] 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.

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

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

[38] 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.

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

[40] 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.

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

[42] H. Cornelius, “High-performance computing on intel architecture,” 2002. Intel Corp.

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

[44] “Personal communication with doug burger and steve keckler,” 2002.

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

[46] 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.

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

[48] 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.

[49] A. Schwerin, S. Swanson, and M. Oskin, "Measuring the complexity-effectiveness of future-generation silicon architecture using fpgas: A status report," in *Proceedings of the 3rd Workshop on Complexity-Effective Design*, 2003.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.