# WaveScalar

Steven Swanson Ken Michelson Andrew Schwerin Mark Oskin
Computer Science and Engineering
University of Washington
{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, without sacrificing parallelism.*

*The WaveScalar ISA is designed to run on an intelligent memory system. 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 in a WaveCache, a simple grid of "alu-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 evaluates a simulated implementation based on current technology. Results for the SPEC and Mediabench applications demonstrate that the WaveCache out-performs an aggressively configured superscalar design by 2-7 times, with ample opportunities for future optimizations.*

## 1 Introduction

It is widely accepted that Moore's Law growth in available transistors will continue for the next decade. Recent research [1], however, has demonstrated that simply scaling up 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 realize 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) the 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. In particular, modern superscalar processor designs will not scale, because they are built atop a vast infrastructure of slow broadcast networks, associative searches, complex control logic, and 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 program counter-driven von Neumann model of computation by providing redundant checking mechanisms [2], exploiting compiler technology for limited dataflow-like execution [3], or efficiently exploiting coarse-grained parallelism [5, 4].

We take a different approach called WaveScalar. At its core, WaveScalar is a dataflow instruction set and computing model [6]. Unlike past dataflow work, which focused on maximizing processor utilization, WaveScalar's goal is to minimize communication costs by ridding the processor of long wires and broadcast networks. To this end, it includes a completely decentralized implementation of the "token-store" of traditional dataflow architectures and a distributed execution model.

The key difference between WaveScalar and prior dataflow architectures is that WaveScalar efficiently supports traditional von Neumann-style memory semantics in a dataflow model. Previously, dataflow architectures provided their own style of memory semantics and their own dataflow languages that disallowed side effects, mutable data structures, and many other useful programming constructs [7, 8]. Indeed, a memory ordering scheme that allows a dataflow machine to efficiently execute code written in general purpose, imperative languages (such as C, C++, Fortran, or Java) has eluded researchers for several decades. In Section 3, we present a memory ordering scheme that efficiently enables true dataflow execution of programs written in *any* language.

Solving the memory ordering problem without resorting to a von Neumann-like execution model allows us to build a completely decentralized dataflow processor that eliminates all the large hardware structures that make superscalars non-scalable. Other recent attempts to build scalable processors, such as TRIPS [3, 9] and Raw [10], have extended the von Neumann paradigm in novel ways, but they still rely on a program counter to sequence program execution and memory access, limiting the amount of parallelism they can reveal. WaveScalar completely abandons the program counter and linear von Neumann execution.

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 also 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

cache-like hardware structures.

This paper is an initial study of the WaveScalar ISA and an example WaveCache architecture. It makes four principle contributions:

- WaveScalar, the first dataflow instruction set to provide total load/store ordering memory semantics which allows efficient execution of programs written in conventional, imperative programming languages.
- An efficient, fully distributed dataflow tag management scheme that is under compiler control.
- An architecture, the WaveCache, that executes WaveScalar programs.
- A quantification of the WaveCache's potential performance relative to an aggressive out-of-order superscalar.

In the next section, we motivate the WaveScalar model by examining three key unsolved challenges with superscalar designs. Sections 3 and 4 describe the WaveScalar instruction set and the WaveCache, respectively. In Section 5, we present a detailed example of the WaveCache's operation that ties together all the WaveScalar concepts. Section 6 presents an initial evaluation of our WaveCache design, and in Section 7, we discuss future work 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. We discuss three: (1) their inherent complexity makes efficient implementation a daunting challenge, (2) they ignore an important source of locality in instruction streams, and (3) their execution model centers around instruction fetch, an intrinsic serialization point.

### 2.1 Complexity

As features and cycle times shrink, the hardware structures that form the core of superscalar processors (register files, issue windows, and scheduling logic) become extremely expensive to access. Consequently, clock speed decreases and/or pipeline depth increases. Indeed, industry recognizes that building ever-larger superscalars as transistor budgets expand can be impractical, because of the processor scaling wall. Many manufacturers are turning to larger caches and chip multiprocessors to convert additional transistors into increased performance without impacting cycle time.

To squeeze maximum performance from each core, architects constantly add new algorithms and structures to designs. Each new mechanism, optimization, or predictor adds additional complexity and makes verification time an ever increasing cost in processor design. Verification already consumes 40% of project resources on complex designs [11], and verification costs are increasing.

### 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. A processor could take advantage of this predictability to reduce the complexity of its communication system (i.e., register files and bypass networks) and reduce communication costs.

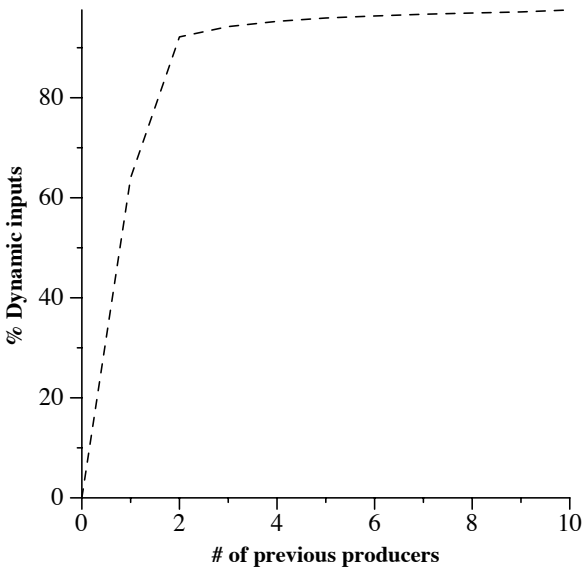


Figure 1: **Dataflow Locality.** The graph quantifies the amount of dataflow locality present in the SPEC2000 integer benchmark suite. A point, $(x, y)$, means that $y\%$ of dynamic instruction inputs are produced by one of their last $x$ producers. For instance, when an instruction reads a value from the register file, there is a 92% chance that it came from one of the last two producers of that value.

Dataflow locality exists, because data communication patterns among static instructions are predictable. There are two independent, but complimentary, types of dataflow locality, static and dynamic. Static dataflow locality exists, because, in the absence of control, the producers and consumers of register values are precisely known. Within a basic block and between basic blocks that are not control dependent (e.g., the basic blocks before and after an If-Then-Else) the data communication patterns are completely static and, therefore, completely predictable. Dynamic dataflow locality arises from branch predictability. If a branch is highly predictable, almost always taken, for instance, then the static instructions before the branch frequently communicate with instructions on the taken path and rarely communicate with instructions on the not-taken path.

Figure 1 demonstrates the combined effect of static and dynamic dataflow locality. The data show that the vast majority of operand communication is highly predictable. For instance, 92% of source operands come from one of their two most recent producer instructions. Such high rates of predictability suggest that current processor communication systems are over-general, because they provide instructions with fast access to many more register values than needed. If the processor could exploit dataflow locality to ensure that necessary inputs were usually close at hand (at the expense of other potential inputs being farther away), they could reduce the average cost of communication.

Instead of simply ignoring dataflow locality, however, superscalars destroy it in their search for parallelism. Register renaming removes false dependencies, enables dynamic loop unrolling, and exposes a large amount of dynamic ILP for the superscalar core to exploit. However, it destroys dataflow locality. 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. This results in 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 communication pattern, destroys that predictability by renaming, and then compensates by using broadcast communication in the register file and the bypass 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.

Several industrial designs, such as partitioned superscalars like the Alpha 21264 [12], some VLIW machines [13, 14], and several research designs [3, 15, 16], have addressed this problem with clustering or other techniques and exploit dataflow locality to a limited degree. But none of these approaches make full use of it, because they still include large forwarding networks and register files. In Section 3, we present an execution model and architecture built expressly to exploit the temporal, spatial, and dataflow locality that exists in instruction and data streams.

### 2.3 The von Neumann model: serial computing

The von Neumann model of computation is very simple. It has three key components: A program stored in memory, a global memory for data storage, and a program counter that guides execution through the stored program. At each step, the processor loads the instruction at the program counter, executes it (possibly updating main memory), and updates the program counter to point to the next instruction (possibly subject to branch instructions).

Two serialization points constrain the von Neumann model and, therefore, superscalar processors: The first arises as the processor, guided by the program counter and control instructions, assembles a linear sequence of operations for execution. The second serialization point is at the memory interface where memory operations must complete (or appear to complete) in order to guarantee load-store ordering. The elegance and simplicity of the model are striking, but the price is steep. Instruction fetch introduces a control dependence between each instruction and the next and serves little purpose besides providing the ordering that the memory interface must adhere to. As a result, von Neumann processors are fundamentally sequential; there is no parallelism in the model.

In practice, of course, von Neumann processors do achieve limited parallelism (i.e., IPCs greater than 1), by using several methods. The explicitly parallel instructions sets for VLIW and vector machines enable the compiler to express instruction and data independence statically. Superscalars dynamically examine many instructions in the execution stream simultaneously, violating the sequential ordering when they determine it is safe to do so. In addition, recent work [17] introduces limited amounts of parallelism into the fetch stage by providing multiple fetch and decode units.

Prior work [18, 19, 20, 21] demonstrated that ample instruction level parallelism (ILP) exists within applications, but that the control dependencies that sequential fetch introduces constrain this ILP. Despite tremendous effort over decades of computer architecture research, no processor comes close to exploiting the maximum ILP present in applications, as 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 [18].

## 3 WaveScalar

This section and the two that follow describe the WaveScalar execution model and the WaveCache architecture. The original motivation for WaveScalar was to build a decentralized superscalar processor core. Initially, we examined each piece of a superscalar and tried to design a new, decentralized hardware algorithm for it. By decentralizing everything, we hoped 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. The problem then became how to build a fully decentralized dataflow machine, and WaveScalar the WaveCache are the creative extension of this line of inquiry.

### 3.1 Dataflow

Dataflow machines are perhaps the best studied alternative to von Neumann processors. The first dataflow architectures [6, 22] appeared in the mid to late 70's, and in the late 80's and early 90's there was a notable revival [23, 24, 25, 26, 27, 28]. Dataflow computers execute programs according to the dataflow firing rule, which stipulates that an instruction may execute at any time after its operands are available. 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.

There have been two significant problems in the development of dataflow as a general purpose computing technology. The first is that 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. [29] articulated this difficulty as a cost/benefit problem and argued that dataflow machines cannot keep sufficient data near the processor to support substantial parallelism. While the arguments were sound, they were based on the assumption that processing elements are expensive and that getting fast memory near the processing elements is difficult. Current technology allows us to build thousands of processing elements on a die and surround each with a small amount of fast storage. As a result, these arguments are no longer applicable.

The second stumbling block was dataflow's inability to efficiently provide total load/store ordering, the memory model assumed by most programming languages. To avoid this problem dataflow researchers resorted to special dataflow languages [30, 31, 32, 33, 34, 35, 36]. While these languages excelled at expressing parallelism that dataflow machines could exploit, they were impractical, because they also disallowed side effects, mutable data structures, and many other programming constructs that are ubiquitous in languages like C, C++, and Java [7, 8].

### 3.2 The WaveScalar ISA

We propose a new approach to dataflow computing, *WaveScalar*, that provides load/store ordering and addresses the problems discussed in Section 2. 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. In practice, since placing a functional unit at each word of instruction memory is impractical, an intelligent instruction cache, a *WaveCache*, holds the current working set of instructions and executes them in place.

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 [22, 37, 38, 39, 40].

**Dataflow instructions** Dataflow machines must convert control dependencies into data dependencies. To accomplish this, they explicitly send data values to the instructions that need them instead of broadcasting them via the register file. 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 $\phi$, instruction [41]. These instructions take two input values and a boolean selector input and, depending on the selector, produce one of the inputs on their output. $\phi$ instructions are analogous to conditional moves and provide a form of predication. They are desirable because they remove the selector input from the critical path of some computations and therefore increase parallelism. They are also wasteful because they discard the unselected input.

The alternative is a conditional split, or $\phi^{-1}$ [39] instruction. The $\phi^{-1}$ instruction takes an input value and a boolean output selector. It directs the input to one of two possible outputs depending on the selector value, effectively steering data values to the instructions that should receive them. These instructions correspond most directly with traditional branch instructions, and they are required for implementing loops.

The WaveScalar ISA supports both types of instructions, but our toolchain currently only supports $\phi^{-1}$ instructions.

**Waves** The WaveScalar compiler breaks the control flow graph of an application into pieces called *waves*. Conceptually, a WaveScalar processor executes a wave at a time. The key properties of a wave are that (1) each time it executes, each instruction in the wave executes at most once, (2) the instructions in the wave are partially ordered (there are no loops), and (3) control can only enter at a single point. These properties allow the compiler to reason about memory ordering within a wave.

Formally, a wave is a connected, directed acyclic portion of the control flow graph with a single entrance. The WaveScalar compiler (or binary translator in our case) partitions an application into maximal waves and adds several wave management instructions (see below).

Waves are similar to hyper-blocks, but can be larger, because they can contain control flow joins. This reduces the amount of overhead due to wave management and makes parallelism easier to extract. In addition, simple loop unrolling is sufficient for generating large waves, whereas hyper-block generation requires heuristics for basic block selection and extensive code replication [42].

**Wave numbers** A significant source of complexity in WaveScalar is that instructions can operate on several instances of data simultaneously. For example, consider a loop. A traditional out-of-order machine can execute multiple iterations simultaneously, because instruction fetch 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.

Traditional dataflow machines use tags to identify different dynamic instances. In WaveScalar every data value carries a tag. We aggregate tag management across waves and use *wave numbers* to differentiate between dynamic waves. A special instruction, WAVE-ADVANCE, manages wave numbers. The WAVE-ADVANCE instruction takes a data value as input, increments the wave number and outputs the original data value with the updated wave number. Because WAVE-ADVANCE is such a simple operation, it can be combined with other instructions. For instance, WaveScalar has an ADD-WITH-WAVE-ADVANCE instruction.

At the top of each wave there is a WAVE-ADVANCE node for each of the wave's live input values. These nodes reside at the entrance to the wave and increment the wave numbers for each input value. As they leave the WAVE-ADVANCE instructions, all values have the same wave number, since they all came from the same previous wave. In the case of a loop, the values of one iteration percolate through the loop body, and the back-edges to the top of the loop lead to the WAVE-ADVANCE instructions. These compute the wave number for the next iteration and ensure that each iteration has a different wave number.

A key feature of WaveScalar is that the WAVE-ADVANCE 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 [31]. In the future, we intend to exploit this fact to optimize WaveScalar binaries by creating application-specific tagging schemes.

**Indirect jumps** Modern systems rely upon object linking and shared libraries, and many programs rely upon indirect function calls. Supporting these constructs requires an additional instruction, INDIRECT-SEND, with 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 both 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 function execution. The caller sends the return address to provide the target address for an INDIRECT-SEND that returns the function's result. The address of the called function need not be known at compile time. A very similar mechanism allows for indirect jumps.

**Memory ordering** Traditional imperative languages provide the programmer with a model of memory known as "total load-store ordering." WaveScalar brings load-store ordering to dataflow computing using *wave-ordered memory*. Wave-ordered memory annotates each memory operation with its location in its wave and its ordering relationships (defined by the control flow graph) with other memory operations in the same wave. As the memory operations execute, these annotations travel with the memory requests and allow the memory system to apply the memory operations in the correct order.

To annotate memory instructions, the WaveScalar compiler statically assigns a unique (within a wave) sequence number to each memory operation by traversing the wave's control flow graph in breadth-first order. Within a basic block, memory operations receive consecutive sequence numbers. By as-

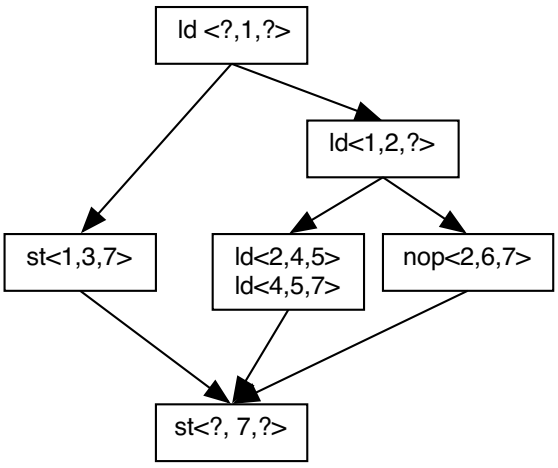


Figure 2: **Annotating memory operations.** A simple wave's control flow graph showing the memory operations in each basic block and their links.

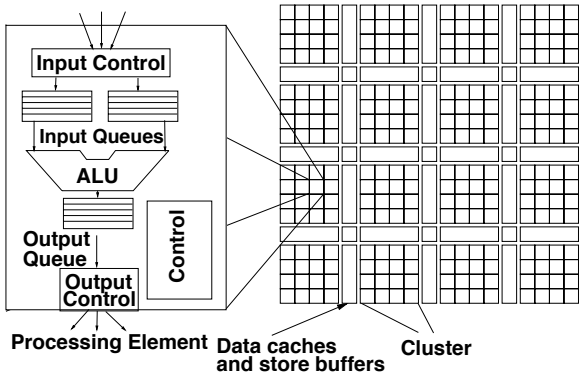


Figure 3: **A Simple WaveCache.** A simple architecture to execute the WaveScalar ISA. The WaveCache combines clusters of processing elements (left) with small data caches and store buffers to form a computing substrate (right).

signing sequence numbers in this way, the compiler ensures that sequence numbers increase along any path through the wave. Next, the compiler labels each memory operation with the sequence numbers of its predecessor and successor memory operations, if they can be uniquely determined. Branches and joins may make this impossible, because they can create multiple successors or predecessors for a single memory operation. 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 $< \mathrm{pred}, \mathrm{this}, \mathrm{succ} >$. Figure 2 provides an example of annotated memory operations in a wave.

The links encode the structure of the wave's control flow graph, and the memory system uses this information to apply operations in the correct order and enforce the load-store ordering the programmer expects. When a memory 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 loads and stores in the correct order to ensure that there are no gaps in the sequence. This is possible because the current wave number in combination with the memory instruction's sequence number totally orders the memory operations through any traversal of a wave, and, by extension, an application. The memory system uses the predecessor and successor information to detect gaps in the sequence. The memory system can be sure that a gap does not exist, if, for each memory operation, $M$, in the sequence, either $M$'s *succ* number matches the sequence number of its next operation, or $M$'s *succ* is '?' and the next operation's *pred* field matches $M$'s sequence number.

To ensure that gap detection is always possible, we must enforce the *no gap rule*: 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 a violation of the rule occurs, the compiler adds a MEMORY-NOP instructions to remove the ambiguity. These instructions participate in memory ordering but otherwise have no effect. In our experiments, MEMORY-NOP's are rare (fewer than 3% of instructions).

Wave-ordered memory is the key to efficiently executing programs written in conventional languages. It allows WaveScalar to separate memory ordering from control flow by succinctly annotating the memory operations with information about their location in the control flow graph. The processing elements are freed from managing 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 system can use this summary in a variety of ways. Currently, we assume a wave-ordered store buffer. Alternatively, a speculative memory system [43, 44, 45] could use the ordering to detect misspeculations.

The most exciting aspect of wave-ordered memory is the incorporation of instruction order into the instruction set as a first-class entity. However, the above scheme for assigning sequence information only encodes information about dependencies between memory operations. It is possible to devise a more general scheme that also expresses independence among memory operations (i.e., that two memory operation can proceed in any order). An ISA that incorporates such a scheme could use memory aliasing information from the compiler to expose large amount of memory parallelism to the hardware. Designing such an ISA and building a suitable compiler are the subject of ongoing work.

## 4 The WaveCache: a WaveScalar processor

In this section, we outline the design of a small WaveCache that could be built within the next 5-10 years to execute WaveScalar binaries (Figure 3). The WaveCache is a grid of approximately 2K processing elements (PEs) arranged into clusters of 16. Each PE contains logic to control instruction placement and execution, input and output queues for instruction operands, communication logic, and a functional unit.

Each PE contains buffering and storage for 8 different instructions, bringing the total WaveCache capacity to 16 thousand instructions – equivalent to a 64KB instruction cache in a modern RISC machine. The input queues for each input require only one write and one read port and as few as 2 entries per instruction (see Section 6.7), or 16 entries total. The input queues are indexed relative to the current wave and a small, multi-ported RAM holds full-empty bits for each entry in the input queues. Matching logic accesses and updates the bits as new inputs arrive, obviating the need for content addressable memories.

In addition to the instruction operand queues, the Wave-

Cache contains a store buffer and a traditional L1 data cache for each 4 clusters of PEs. The caches access DRAM through a conventional, unified, non-intelligent L2 cache. Total storage in the WaveCache is close to 4MB when input and output queues and the L1 data caches are accounted for.

Within a cluster, the processing elements communicate via a set of shared buses. Tiles within the same cluster receive results at the end of the clock cycle during which they were computed. Cluster size is one of the key architectural parameters of the WaveCache. Larger clusters require more wires and more area for intra-cluster communication, while smaller clusters increase inter-cluster communication costs. However, data in Section 6.5 demonstrate that even with singleton clusters, WaveCache performance is still very good.

For inter-cluster communication, the WaveCache uses a dynamically routed on-chip network. Each "hop" in the network crosses one cluster and takes a single cycle. In Section 6.5, we demonstrate that contention on the network will be minimal.

During execution, each instruction is bound to a processing element, where it remains for possibly millions of executions. Good instruction placement is paramount for optimal performance, because communication latency depends on instruction placement.

Our initial design uses a distributed store buffer. Each set of 4 processing element clusters contains a store buffer shared among 64 functional units. Each dynamic wave is bound to one of these, and all the memory requests for the wave go to that store buffer. As a store buffer completes, it signals the store buffer for the next wave to proceed. This scheme allows the store buffer to be logically centralized but to migrate around the WaveCache as needed.

The remaining difficulty is how to assign store buffers to waves. We propose two different solutions. The first is to add an instruction, MEM-COORDINATE, that acquires a store buffer on behalf of its wave and passes the name of the store buffer to the memory instructions in the wave. This forces a dependency between the MEM-COORDINATE and all memory operations, slightly reducing parallelism.

The second solution uses a hash table kept in main memory that maps wave numbers to store buffers. Memory instructions then send their requests to the nearest store buffer. The store buffer accesses the map to determine where the message should go. If the current wave does not yet have a store buffer, the store buffer uses the cache coherence system to get exclusive access to the wave's entry in the hash, fills it with its own location, and processes the request. If the map already has an entry for the current wave, it forwards the message to the appropriate store buffer. While this scheme does not reduce parallelism in the dataflow graph, it places additional pressure on the memory system and adds complexity to the store buffer.

## 5 Example

In this section, we describe the creation and execution of a WaveScalar executable in detail. To motivate the discussion, we use the simple C program fragment shown in Figure 4.

### 5.1 Compilation

Compiling an executable for a WaveScalar processor is much the same as compiling for a conventional architecture. Figure 5 shows the relationship between the traditional RISC assembly for the code fragment (a) and the WaveScalar equivalent (b). In this example, the WaveScalar program contains all of the instructions in the RISC version. After the traditional

```
function s(char in[10], char out[10]) {
     i = 0;
     j = 0;
     do {
          int t = in[i];
          if (t) {
               out[j] = t;
               j++;
          }
          i++;
     } while (i < 10);
     // no more uses of i
     // no more uses of in
}
```

Figure 4: **Example code fragment:** This simple loop copies `in` into `out`, ignoring zeros.

compiler stages (parsing and optimization), the transformation proceeds in four stages.

First, the compiler decomposes the dataflow graph into waves. In this case, the loop body is a single wave. Any code before or after the loop would be in separate waves. In WaveScalar executables all inner-loop bodies are a single wave.

Second, the compiler inserts MEMORY-NOP instructions and assigns sequence numbers to the memory operations filling in the predecessor and successor fields. In this example, MEMORY-NOP is required to make the memory ordering unambiguous. If the MEMORY-NOP were not present, there would be a direct path from the load in one iteration to the load in the next (if control bypassed the store), violating the no gap rule. In essence, the MEMORY-NOP informs the memory system when the store is not going to occur. In the example, either the store in row 7 or the MEMORY-NOP in row 6 can follow the load, so a '?' fills the successor position in the load's link. The load's predecessor is also a '?', because the previous memory operation may be just before the loop or at the tail of the loop body.

Third, the compiler inserts WAVE-ADVANCE instructions at the top (row 1) of each wave for each live-in. In the example, the loop body has four live-in values, but since `i` and `in` are not used after the loop body, only two WAVE-ADVANCE instructions (row 9) are required to start the wave that follows the loop.

Finally, the compiler converts the branch instructions into predicates (the two instructions in row 4) and inserts $\phi^{-1}$ instructions to steer data values to the appropriate instructions. The branch within the loop must steer three values (`j`, `out`, and `t`), so it requires three $\phi^{-1}$ instructions (row 5); the backward branch to the top of the loop body requires four (row 8).

Once these transformations are complete, the graph contains all the information needed to execute the program correctly. At this point, the compiler could apply additional, WaveScalar-specific optimizations. Implementing such optimizations are the subject of future work.

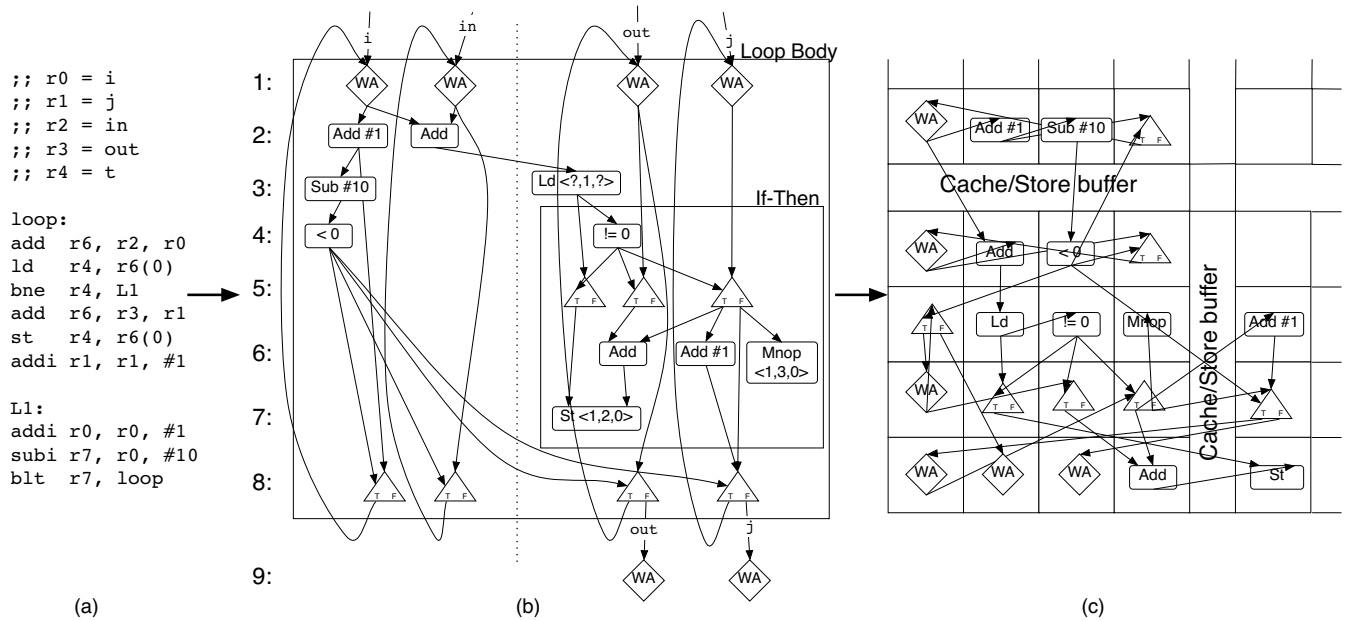


Figure 5: **WaveScalar example.** The RISC assembly (left) for the program fragment in Figure 4, the WaveScalar version (center), and the WaveScalar version mapped onto a small WaveCache (right). To clarify the discussion in the text, the numbers in the column label rows of instruction, and the vertical, dashed line divides the graph into two parts.

### 5.2 Encoding

WaveScalar binaries are larger than traditional RISC binaries for two reasons. First, WaveScalar instructions are larger than RISC instructions. Second, more instructions are needed for data steering ($\phi^{-1}$), wave management (WAVE-ADVANCE), and memory ordering (MEMORY-NOP).

Each WaveScalar instruction contains an opcode and a list of targets for each of the instruction's outputs. A target contains the address of the destination instruction and an input number to designate which input should receive the output values. WaveScalar instructions have up to three inputs and two outputs. In practice, the number of targets for each output will be limited, but currently, we allow an unbounded number of targets. Although encoding all this information requires more than 32 bits, the size of individual instructions is of little importance, because they are loaded once and executed many times.

The additional instructions WaveScalar requires could potentially be more troublesome, since more instructions lead to more contention for the WaveCache's functional units. We address this question in Section 6.4.

### 5.3 Loading, finding, and replacing instructions

Execution in the WaveCache is analogous to execution on a superscalar that can only execute instructions present in the I-Cache. To initiate execution on a conventional superscalar the operating system sets the PC to the beginning of the program. An I-Cache miss results and loads part of the executable (a cache line), which executes and causes more cache misses.

In the WaveCache, an instruction in the operating system sends a message to the address of the first instruction in a program. Since the instruction is not present in the WaveCache, a miss occurs, and the WaveCache loads the instruction into a functional unit. When placing a new instruction into a functional unit, the WaveCache rewrites the targets of its outputs to point to their location in the WaveCache instead of their locations in memory. In the case of the first instruction, however, none of the destinations are present, so the targets are set to a special "not-loaded" location. Any messages sent to the "not-loaded" location result in additional misses. Therefore, when the missing instruction arrives and fires, it generates additional misses that result in more instructions being loaded.

In the WaveCache, a miss is a heavyweight event, because all of the instruction's state (input queues, etc.) must be brought onto the chip instead of just the instruction itself. In addition, 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 (overwriting their destination fields with "not-loaded"). To reduce the cost of removal, instructions mark themselves as "idle" if their queues are empty and they have not executed for some time. Idle instructions have no state to write back to memory, so other instructions can cheaply replace them. At the same time, if a message arrives before they are displaced, they can spring back into action immediately.

Eventually, the working set of instructions stabilizes. Figure 5(c) shows one possible mapping of the example onto a small WaveCache. Once the instructions are in place, they execute in place by passing values within the grid.

### 5.4 Execution

As the loop in Figure 5(b) begins to execute, values arrive along the input arcs at the WAVE-ADVANCE instructions at the top of the loop. We assume that `i` and `j` have already been initialized to 0; although it may not be the case in practice, that the inputs arrive at the WAVE-ADVANCE instructions simultaneously and that the dynamic wave just before the loop is wave 0.

We first consider the subgraph to the left of the vertical

dotted line. On the first cycle, `i` and `in` pass through their WAVE-ADVANCE instructions unchanged except that their wave numbers are incremented to 1. The resulting value of `in` passes directly to one of the $\phi^{-1}$ instructions in row 8, where it waits for the control input to arrive. On the next cycle, "ADD #1" increments the value of `i` and passes the result to a second $\phi^{-1}$ instruction. At the same time, "ADD" computes the load address and sends it to the load instruction. We will discuss this value's fate shortly. "SUB #10" fires next and triggers "$< 0$", which in turns provides the control input ("true") to the $\phi^{-1}$ instructions in row 8. These fire and send values from their "true" output to the WAVE-ADVANCE at the top of the loop. The second iteration begins with `i`$= 1$ and `in` unchanged; these values pass through WAVE-ADVANCE instructions, and their wave numbers rise to 2. The next 8 iterations proceed in the same fashion. On the 10th and final iteration, "$< 0$" produces "false," and since the $\phi^{-1}$ instructions for `i` and `in` have no consumers for their "false" outputs, they produce no values at all.

The right-hand side proceeds in a similar manner. After the WAVE-ADVANCE instructions increment their wave numbers, `j` and `out` flow directly to two $\phi^{-1}$ instructions where they await the arrival of the control input from "!=0." Simultaneously, once the load has an input available (sent from the left-hand side of the graph), it sends a request to the memory system and awaits a response. When the input arrives, the load forwards it to "!=0" and a $\phi^{-1}$ in row 5. Once one of the $\phi^{-1}$ instructions has both its data value and its control input, it fires and, depending on whether the value returned by the load was equal to zero, sends the value on either its "true" or "false" output. In the case of a non-zero value, both ADD instructions in row 6 fire. One performs the address computation, and the other increments `j` and passes it to the $\phi^{-1}$ in row 8. On the next cycle, the store fires and sends its link, the address, and the data value to the memory system. If the load value is equal to zero, only one $\phi^{-1}$ (the rightmost) produces an output, and as a result, the MEMORY-NOP fires and sends its link to the memory system. Execution continues until the final iteration, when the final value of `j` passes out of the loop along with the value of `out`.

Note that the left-hand side of the WaveScalar graph is independent of the right-hand side (edges *only* cross the dashed line from left to right). This means that the computation of the loop index, `i`, and the load addresses can potentially proceed much more quickly than the rest of the loop. We frequently observe this behavior in practice. In this case the values from the "ADD" in row 2 and the "$< 0$" in row 4 accumulate at the inputs to the LD and the two right-hand $\phi^{-1}$ instructions, respectively.

This has two competing effects. First, it enables the load to fire requests to the memory system as quickly as possible, allowing for greater throughput. Second, it could overflow the input queue at the load, forcing values to spill to memory, increasing memory traffic and reducing throughput (see Section 6.7). Finding the correct balance between these two effects will involve providing an intelligent back-pressure mechanism and is a subject of future work.

### 5.5 Termination

Terminating a program running in the WaveCache is more complicated than terminating a program on a von Neumann processor. Instead of descheduling the process and releasing any resources it holds, the operating system must forcibly remove any instructions remaining in the WaveCache that belong to the process. While this "big hammer" capability is necessary to prevent malicious programs from consuming resources unchecked, a gentler, more fine grained mechanism is also desirable and is the subject of future work.

## 6 Results

In this section, we explore the performance of the WaveScalar ISA and the WaveCache. We investigate seven aspects of execution: WaveCache performance relative to a superscalar and the TRIPS [3] processor; the overhead due to WaveScalar instructions; and the effects of cluster size, cache replacement, and input queue size on performance, as well as the potential effectiveness of control and memory speculation. These results demonstrate the potential for good WaveScalar performance and highlight fruitful areas for future investigation.

### 6.1 Methodology

Our baseline WaveCache configuration is the system described in Section 4 with 16 processing element per cluster, unbounded input queues, and perfect L1 data caches. The baseline configuration executes nothing speculatively. For some studies, we bound input queue size, vary the size of the WaveCache, and implement speculation mechanisms. 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 algorithm [46] to improve layout. We also model an optimization that allows store addresses to be sent to the memory system as soon as they are available, possibly before the corresponding data. This allows loads to other accesses to return values more quickly.

For comparison, we use a high-performance superscalar machine with 15 pipeline stages; a 16-wide, out-of-order processing core; 1024-physical registers; and a 1024-entry issue window with oldest-instruction-first scheduling. The core uses an aggressively pipelined issue window and register file similar to that described in [47] to reduce critical scheduling/wake-up loop delays. The core also includes a gshare branch predictor [48], store buffer, and perfect (i.e., 16-ported) cache memory system. Neither the WaveCache nor the superscalar speculate on any memory dependences, although we relax this constraint for the comparison in Section 6.8. Since the pipeline is not partitioned, 15 stages is aggressive given the register file and issue window sizes and the width of the machine.

To perform a fair comparison to the superscalar design and to leverage existing compiler technology, 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 the von Neumann to the dataflow computing model. We compile a set of benchmarks using the Compaq cc (v5.9) compiler on Tru64 Unix, using the `-O4` and `-unroll 16` flags. The benchmarks are *vpr*, *twolf*, and *mcf* from SPECint2000 [49]; *equake* and *art* from SPECfp2000; *adpcm* and *mpeg2encode* from mediabench [50]; and *fft*, a kernel from Numerical Recipes in C [51]. We chose these benchmarks because they provide a variety of application types and can be processed by our binary translator. We use *gprof* to identify the functions that account for 95% of execution time and convert them into the WaveScalar ISA using our binary translator. Finally, we simulate a segment of execution equivalent to 100 million dynamic Alpha instructions (or to completion) for each benchmark.

We report the results in terms of *Alpha-equivalent instructions per cycle* (AIPC). For the WaveCache measurements we carefully distinguish between instructions from the original Alpha binary and those the Alpha-to-WaveScalar bi-

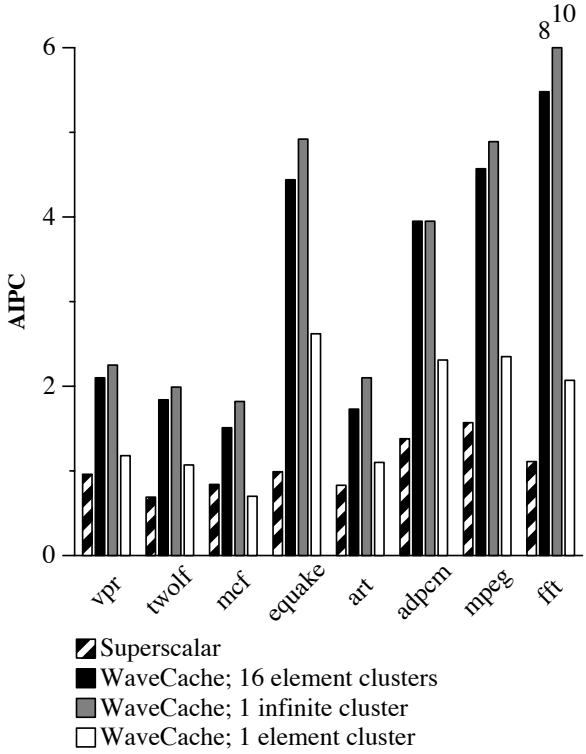


Figure 6: **Superscalar vs. WaveCache.** We evaluate each application on the superscalar and the WaveCache with 16 element element clusters, a single, infinite cluster, and one element clusters.

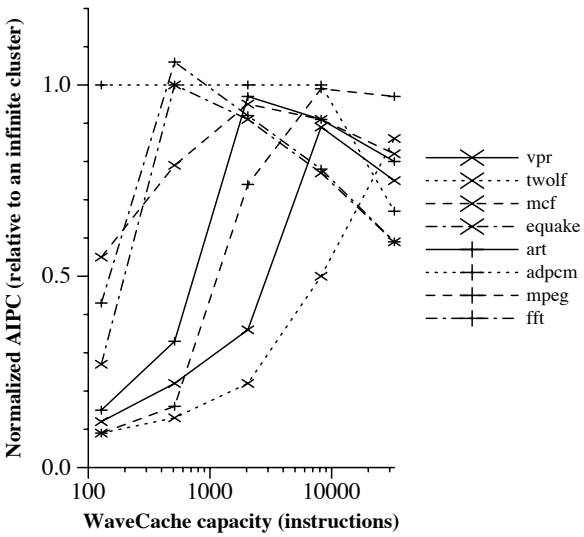


Figure 7: **WaveCache capacity.** The curves depict performance for each benchmark as the capacity of the WaveCache increases. Performance peaks and then decreases for larger sizes, demonstrating the importance of dynamic instruction placement.

nary rewriter adds ($\phi^{-1}$, WAVE-ADVANCE, etc.) but do ***not*** include the latter in any of the throughput measurements. We use AIPC, because it fairly compares the amount of application-level work performed by each processor.

The performance we report is a conservative estimate of the WaveCache's real potential. We rely on a binary translator to produce WaveScalar binaries, and a compiler built to optimize for the WaveScalar ISA and target the WaveCache implementation would only improve performance.

### 6.2 Comparison to the superscalar

Figure 6 compares the WaveCache to the superscalar. The WaveCache with 16 processing elements per cluster outperforms the superscalar by a factor of 3.1 on average. For highly loop parallel applications such as *equake* and *fft*, the WaveCache is 4-7 times faster than the superscalar. The WaveCache outperforms the superscalar by such substantial margins, because the WaveCache does not introduce false control dependencies that artificially constrain ILP. The increase in IPC belies an even greater improvement in overall performance, since it is unlikely that the superscalar we simulate could match the WaveCache's cycle time.

### 6.3 Comparison to TRIPS

It is interesting to compare the WaveCache's performance to that of the TRIPS [3, 9] processor, a recent VLIW-dataflow hybrid. TRIPS bundles hyperblocks of VLIW instructions together vertically and describes their dependencies explicitly instead of implicitly through registers. Within a hyperblock, instructions fire according to the dataflow firing rule, while communication between hyperblocks occurs through a register file. At its core, TRIPS remains a von Neumann architecture, because the program counter still determines the sequence of hyperblocks the processor executes.

Our memory speculation study (Section 6.8.2) and the perfect-cache "D-mode" results of recent TRIPS work [9] use nearly identical assumptions (a perfect memory system with memory disambiguation). Both studies also ignore "overhead" instructions and count only RISC-like instructions on the correct path. For these reasons, a cross-study comparison of IPC is interesting, although not perfect, because the two studies use different compilers (the Trimaran compiler for TRIPS and the DEC/cc compiler combined with a binary rewriter for WaveScalar). One significant difference between the two studies is the amount of bypassing logic. The WaveCache allows back-to-back execution of instructions in the same cluster, perhaps leading to a longer cycle time, while TRIPS incurs no delay for bypassing to the same functional unit and 1/2 cycle of delay per node hop to access remote functional units.

For the applications that the two studies have in common (*adpcm*, *art*, *equake*, *twolf*, *mcf*, and *mpeg*), the baseline WaveCache configuration achieves a factor of 2.5 more IPC than the baseline TRIPS architecture. However, a modified TRIPS design with 16 node clusters and full bypassing that matches the WaveCache reduces the performance gap to 90% [52]. Interestingly, the performance improvements are not uniform and suggest improvements can be made to both architectures. TRIPS is nearly twice as fast for *art* and 42% faster on *twolf*. We speculate that TRIPS outperforms the WaveCache on these applications due to poor instruction placement. A badly chosen placement in the WaveCache spreads instruction dependencies across clusters, contributing additional latency. For all other applications the WaveCache outperforms TRIPS because of its pure dataflow execution model. This, combined with the execution of waves instead of hyperblocks (which are smaller), enables execution across larger regions of the application, unlocking IPC that is hid-

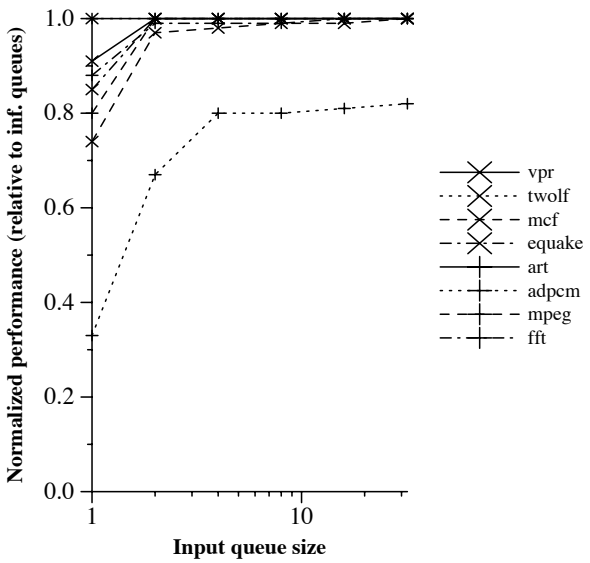


Figure 8: **Input Queue Size.** Queue size has a relatively small effect on WaveCache performance.

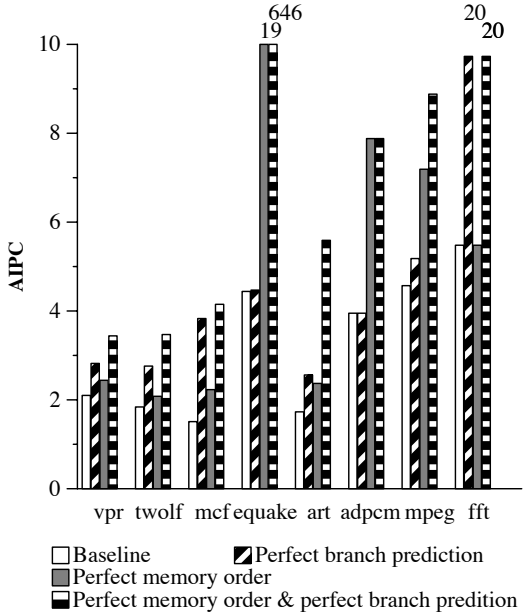


Figure 9: **WaveCache Speculation.** Comparison of the baseline WaveCache configuration with a variety of speculation schemes.

den by the program counter in von Neumann machines. This suggests that the memory ordering strategy introduced in this paper would benefit a TRIPS-like processor.

### 6.4 Overhead

For our benchmarks, static instruction count overhead varies from 20% to 140%. Although this overhead is substantial, the effect on performance is small. Most of the added instructions execute in parallel (for instance the WAVE-ADVANCE instructions at the top of the wave), and removing their latency improves performance by only 11% on average.

### 6.5 Cluster size

One of the WaveCache's goals is to communicate data efficiently and to achieve high performance without resorting to long wires. We now examine this aspect of the WaveCache in detail. Recall that communication takes a single cycle within a cluster. Since larger clusters require longer wires and, therefore, a slower clock, cluster size is a key parameter. Figure 6 shows performance results for a configuration with a single, infinite cluster as an upper bound on WaveCache performance.

Overall, the performance of 16-processing-element clusters is 90% of the infinite case. Sixteen-element clusters strike a nice balance between smaller, 4-element clusters (73% of infinite) and larger, 64-elements clusters (97%). Sixteen-element clusters also do a fine job of capturing dataflow locality. For each benchmark, less than 15% of values leave their cluster of origin, and less than 10% must cross more than one cluster to reach their destination. Because so few messages must leave their cluster of origin, fewer than 5 data messages (on average) are in flight on the inter-cluster network each cycle. The additional protocol traffic (cache coherence, etc.) that we do not yet model increases the load, but given this initial result, we do not expect contention in the interconnect to be a bottleneck.

The figure also shows performance with isolated processing elements. Using singleton clusters reduces performance by 51%. While this may seem like a dramatic drop, a single-element cluster WaveCache still outperforms a 15-stage superscalar by an average of 72%. Tiny clusters also reduce wire length and increase the potential clock rate dramatically.

### 6.6 Cache replacement

To measure the effects of capacity on WaveCache performance, we vary the cache capacity from a single processing element cluster (128 instructions) to a 16-by-16 array of clusters (32K instructions). In each case, every instruction is statically assigned to a cluster and replacement is done on a per-cluster basis according to an LRU algorithm. We assume 32 cycles to read the instruction state from memory and between 1 and 24 cycles to transmit the data across the cache, depending on the cache size.

Figure 7 shows the effect of cache misses on performance. For all the benchmarks except *adpcm*, which has a very low miss rate, performance increases quickly as the miss rate declines. For four applications (*mcf*, *equake*, *fft*, and *mpeg*), performance decreases as the cache size increases past this point, because spreading *all* the instructions in an application across a large WaveCache is inefficient. Allowing the cache replacement policy to collect the working set of instructions in a smaller physical space, thereby reducing communication latency and increasing performance, is better. The ability to exploit dynamic instruction placement sets the WaveCache apart from other "grid" architectures like RAW [10], TRIPS [9], and Nanofabrics [40]. By letting the current needs of the program guide instruction placement, the WaveCache can aggressively exploit dataflow locality and achieve higher performance. Dynamic instruction placement and optimization is one of our highest priorities for future work.

### 6.7 Input queue size

The WaveScalar model assumes that each instruction can buffer an unlimited number of input values and perform wave number matching among them. In reality, limited storage is available at each processing element. Each input queue in the WaveCache 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. To ensure that in-memory matching is rare, we use a simple prefetching mechanism that brings values back into the queue as space becomes available. When a space becomes available at the end of an input queue, the prefetch unit attempts to load the value with the next wave number (i.e., the largest wave number in the queue plus 1).

Figure 8 shows the effect of queue size on performance. For all but the smallest queue sizes, performance is largely unaffected by queue size. This is encouraging because more instructions can fit into the WaveCache, if smaller queues are used. One application, *adpcm*, shows a 20% slowdown with even 32 entry queues. This is a manifestation of the well-known "parallelism explosion" problem with dataflow architectures [53]. In the future, we plan to address this issue by adding a back pressure mechanism to keep the required queue size small.

### 6.8 Control and memory speculation

Speculation is required for high performance in superscalar designs [54]. The WaveCache can also benefit from speculation but does not require it for high performance. We investigate the limits of both control and memory independence prediction in the WaveCache.

#### 6.8.1 Control speculation

In the WaveCache, perfect branch prediction means that $\phi^{-1}$ instructions steer values to the correct output without waiting for the selector input. Perfect branch prediction increases the performance of the WaveCache by 47% on average (Figure 9). This represents a significant opportunity to improve performance, since the WaveCache currently uses no branch prediction at all.

#### 6.8.2 Memory speculation

In many cases, 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.

Figure 9 shows that memory disambiguation gives a substantial boost to WaveCache performance, increasing it by an average of 62%. With perfect disambiguation, WaveScalar out-performs the superscalar by an average of 123%. Perfect disambiguation combined with perfect branch prediction yields a 340% improvement over the baseline WaveCache. The results for *equake* are especially interesting, since they demonstrate that, with speculation, WaveScalar can automatically parallelize applications to great effect.

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. We have also shown that the WaveCache does not require speculation to achieve high performance, as von Neumann processors do.

## 7 Conclusion

In this paper, we have presented WaveScalar, a new dataflow 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. By dividing the program into waves WaveScalar provides decentralized, inexpensive, software-controlled tag management. 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 2-7 performance increase in our study of the SPEC and mediabench applications when compared to an aggressively configured superscalar processor. It achieves these gains in a communication-scalable architecture, without speculation.

We have only begun to study WaveScalar architectures. Many exciting challenges remain including handling interrupts, I/O, and other operating system issues. WaveScalar also presents some tantalizing opportunities. For example, given its large number of processing elements, the WaveCache should efficiently execute workloads with enormous inherent parallelism. How do we realize this potential? Likely approaches include extending the model to include threads and other mechanisms for expressing the parallelism present in the application as well as developing programming paradigms that express as much parallelism as possible.


### Acknowledgements

We would like to thank Susan Eggers, Mike Swift, Andrew Petersen, Patrick Crowley, Timothy Sherwood, Mark Horowitz, Todd Austin, and Steve Keckler for providing comments and suggestions on early versions of this work. We would also like to thank Ramdass Nagarajan and Steve Keckler for providing the TRIPS comparison data in Section 6.3. This work is funded in part by NSF CAREER Award ACR-0133188, NSF Doctorate and Intel Fellowships (Swanson), ARCS Fellowship (Schwerin), and Mary Gates Fellowship (Michelson).


### 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 *International Symposium on Computer Architecture*, 2000.

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

[3] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler, "A design space evaluation of grid processor architectures," in *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*, vol. 30, no. 9, 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 International Symposium on Computer Architecture*, 1975.

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

[8] 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*, vol. 29, no. 9, 1980.

[9] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. Keckler, and C. Moore, "Exploiting ILP, TLP, and DLP with the polymorphous trips architecture," in *International Symposium on Computer Architecture*, 2003.

[10] W. Lee *et al.*, "Space-time scheduling of instruction-level parallelism on a Raw machine," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

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

[12] L. Gwennap, "Digital 21264 sets new standard," *Microprocessor Report*, vol. 10, October 1996.

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

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

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

[16] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *International Symposium on Computer Architecture*, ACM Press, 1997.

[17] P. S. Oberoi and G. S. Sohi, "Parallelism in the front-end," in *Proceedings of the 30th annual international symposium on Computer architecture*, pp. 230–240, ACM Press, 2003.

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

[19] D. W. Wall, "Limits of instruction-level parallelism," in *International Conference on Architectural Support for Programming Languages and Operating System*, 1991.

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

[21] A. A. Nicolau and J. Fisher, "Using an oracle to measure potential parallelism in single instruction stream programs," in *Microprogramming Workshop*, 1981.

[22] A. L. Davis, "The architecture and system method of ddm1: A recursively structured data driven machine," in *International Symposium on Computer Architecture*, 1978.

[23] S. Sakai, y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba, "An architecture of a dataflow single chip processor," in *International Symposium on Computer Architecture*, 1989.

[24] T. Shimada, K. Hiraki, K. Nishida, and S. Sekiguchi, "Evaluation of a prototype data flow processor of the sigma-1 for scientific computations," in *International Symposium on Computer Architecture*, 1986.

[25] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, no. 1, 1985.

[26] M. Kishi, H. Yasuhara, and Y. Kawamura, "Dddp-a distributed data driven processor," in *International Symposium on Computer Architecture*, 1983.

[27] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes, "The epsilon dataflow processor," in *International Symposium on Computer Architecture*, 1989.

[28] G. Papadopoulos and D. Culler, "Monsoon: An explicit token-store architecture," in *International Symposium on Computer Architecture*, 1990.

[29] D. E. Culler, K. E. Schauser, and T. Eicken von, "Two fundamental limits on dataflow multiprocessing," in *IFIP Working Group (Concurrent Systems) Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, 1993.

[30] R. Nikhil, "The parallel programming language Id and its compilation for parallel machines," in *Workshop on Mazzive Paralleism: Hardware, Programming and Applications*, 1990.

[31] Arvind and R. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *IEEE Transactions on Computers*, vol. 39, no. 3, 1990.

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

[33] S. Murer and R. Marti, "The FOOL programming language: Integrating single-assignment and object-oriented paradigms," in *European Workshop on Parallel Computing*, 1992.

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

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

[36] J. R. McGraw, "The VAL language: Description and analysis," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 1, 1982.

[37] J. B. Dennis, "Dataflow supercomputers," in *IEEE Computer*, vol. 13, 1980.

[38] 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 *Programming Language Design and Implementation*, 1990.

[39] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

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

[41] 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, no. 4, 1991.

[42] 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 *International Symposium on Microarchitecture*, 1992.

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

[44] 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*, 1997.

[45] M. Franklin and G. S. Sohi, "ARB: a hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, vol. 45, no. 5, 1996.

[46] 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*, 2002.

[47] 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 *International Symposium on Computer Architecture*, 2002.

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

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

[50] 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*, 1997.

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

[52] "Personal communication with doug burger and steve keckler," 2002-2003.

[53] D. E. Culler and Arvind, "Resource requirements of dataflow programs," in *International Symposium on Computer architecture*, 1988.

[54] S. Swanson, L. McDowell, M. Swift, S. Eggers, and H. Levy, "An evaluation of speculative instruction execution on simultaneous multithreaded processors," *to appear in Transactions on Computer Systems*, 2003.