Area-Performance Trade-offs in Tiled Dataflow Architectures

Steven Swanson Andrew Putnam Martha Mercaldi Ken Michelson Andrew Petersen
Andrew Schwerin Mark Oskin Susan J. Eggers
Computer Science & Engineering
University of Washington
{swanson,putnam,mercaldi,ken,petersen,schwerin,oskin,eggers}@cs.washington.edu

Abstract:

Tiled architectures, such as RAW, SmartMemories, TRIPS, and WaveScalar, promise to address several issues facing conventional processors, including complexity, wire-delay, and performance. The basic premise of these architectures is that larger, higher-performance implementations can be constructed by replicating the basic tile across the chip.

This paper explores the area-performance trade-offs when designing one such tiled architecture, WaveScalar. We use a synthesizable RTL model and cycle-level simulator to perform an area/performance pareto analysis of over 200 WaveScalar processor designs ranging in size from 19mm^2 to 378mm^2 and having a 22 FO4 cycle time. We demonstrate that, for multi-threaded workloads, WaveScalar performance scales almost ideally from 19 to 101mm^2 when optimized for area efficiency and from 44 to 202$\mathrm{mm}^2$when optimized for peak performance. Our analysis reveals that WaveScalar's hierarchical interconnect plays an important role in overall scalability, and that WaveScalar achieves the same (or higher) performance in substantially less area than either an aggressive out-of-order superscalar or Sun's Niagara CMP processor.

Keywords: WaveScalar, Dataflow computing, ASIC, RTL

1 Introduction

To address a set of critical problems in processor design, including design complexity, wire delay, and fabrication reliability, many computer architects are beginning to shift their focus away from today's complex, monolithic, high-performance processors. Instead, they are designing a much simpler processing element (PE) and compensating for its lower individual performance by replicating it across a chip. Examples of these *tiled architectures* include RAW [1], SmartMemories [2], TRIPS [3, 4] and WaveScalar [5]. Their simple PEs decrease both design and verification time, PE replication provides robustness in the face of fabrication errors, and the combination reduces wire delay for both data and control signal transmission. The result is an easily scalable architecture that enables a chip designer to capitalize on future silicon process technologies.

Despite the high-level simplicity and regularity of the tiled structure, good performance on these architectures—more importantly, good performance per unit area—is achievable only if all aspects of the microarchitecture are properly designed. Architects face some of the same design issues in these systems as in conventional, more centralized processors (e.g., ALU mix, cache hierarchy design). However, they also face a new set of issues. For example, should architectures have more tiles to exploit additional parallelism or fewer, more highly utilized and possibly more powerful ones? Where should the various data memories be located, and to what extent should they be partitioned and distributed around the die? Is it more important to devote area to additional processing elements or memories? How should tiles be interconnected?

This paper explores the area-performance trade-offs encountered when designing a tiled architecture. The target architecture for this study is the WaveScalar processor, a tiled dataflow architecture. Like all dataflow architectures, each of WaveScalar's simple PEs execute instructions according to the dataflow firing rule [6]. Data communication between instructions in different PEs is explicitly point-to-point. To reduce data communication costs, instructions that communicate frequently are placed in close proximity [7], and the data networks are organized hierarchically. All major hardware data structures are distributed across the die, including the caches, store buffers and specialized dataflow memories such as the token store.

To accurately conduct an area-performance study, we use two artifacts. The first is a synthesizable Verilog model of the WaveScalar architecture. This model synthesizes with the latest commercial design tools to a fast clock (22 FO4) on the latest available TSMC process (90nm). The Verilog model can execute instructions, but it, like all RTL simulators, is too slow for executing millions of them. For this purpose we wrote a corresponding cycle-level simulator and application development tool-chain.

This paper uses these tools to make two primary contributions. First, it revisits the buildability of dataflow machines using today's denser technology. Second, it explores a wide swath of the area-performance design space of this dataflow architecture.

In particular, we describe the microarchitecture for two key parts of the WaveScalar processor: its most area-intensive components, including those that implement its distributed dataflow capabilities, and the interconnection networks that localize data communication, contributing heavily to performance. This basic architecture can be used to build a continuum of WaveScalar processors with varying performance and area. We detail the area budget for a particular configuration, showing where it is crucial to optimize the design.

Then, to understand a larger design space, we use data from our RTL synthesis to develop an area model that describes the area requirements for a range of designs. We use the resulting model to enumerate a large set of WaveScalar processors that could be built in modern process technology. Our evaluation of these designs, a pareto analysis of the design space, demonstrates that multithreaded WaveScalar processor performance scales almost ideally from 19mm^2 to 101mm^2 of silicon in terms of area efficiency and from 44mm^2 to 202mm^2 in terms of peak performance. Scaling the designs by an additional factor of four yields double performance. We then compare WaveScalar's area efficiency to an aggressive out-of-order superscalar and Sun's Niagara chip multiprocessor and find that WaveScalar achieves sub-

stantially more performance per unit area than either of these designs. Finally, we analyze the effect of processor size on the on-chip interconnect performance, and find that our processor's hierarchical interconnect does an excellent job of minimizing latency by localizing communication, with more than 98% of communication occurring within a single WaveScalar cluster.

This paper begins in Section 2 by describing our experimental infrastructure. Section 3 presents an overview of the WaveScalar microarchitecture and its four main components: (1) the execution tile, i.e., the processing element (PE); (2) an interface to memory that supports imperative languages, which we call a *wave-ordered storebuffer*; (3) a conventional data-cache hierarchy; and (4) a hierarchical interconnect to provide fast communication among these components. The description of the four components is split between Section 3, which contains a basic description of their design and architecture, and an appendix, which contains a detailed example of PE operation. Section 4 presents results of the area-performance analysis. Section 5 describes other tiled-architecture work and Section 6 concludes.

2 Experimental infrastructure

In this section we describe the RTL toolchain and simulation methodology that produced the area-performance results presented throughout this paper.

2.1 The RTL model

Being a tiled dataflow processor, WaveScalar is different enough from conventional von Neumann processors that we cannot draw on past research, existing tools, or industrial experience to understand area and cycle-time requirements. Since these parameters are crucial for determining how well WaveScalar performs and how to partition the silicon resources, we constructed a synthesizable RTL model of the components described in later Sections 3.2 through 3.4.

The synthesizable RTL model is written in Verilog, and targets a 90nm ASIC implementation. Considerable effort was put into designing, and redesigning this Verilog to be both area-efficient and fast. The final clock speed (22 FO4) comes from our fourth major redesign.

The 90nm process is the most current process technology available, so the results presented here should scale well to future technologies. In addition to using a modern process, we performed both front-end and back-end synthesis to get as realistic a model as possible. The model makes extensive use of Synopsys DesignWare IP [8] for critical components such as SRAM controllers, queue controllers, arbiters, and arithmetic units. The design currently uses a single frequency domain and a single voltage domain, but the tiled and hierarchical architecture would lend itself easily to multiple voltage and frequency domains in the future.

ASIC design flow: We used the most up-to-date tools available for Verilog synthesis. Synopsys VCS provided RTL simulation and functional verification of the post-synthesis netlists. Front-end synthesis was done using Synopsys DesignCompiler. Cadence FirstEncounter handled back-end synthesis tasks such as floorplanning, clock-tree synthesis, and place and route [9]. By using back-end synthesis, the area and timing results presented here include realistic physical effects, such as incomplete core utilization and wire delay, that are critical for characterizing design performance in 90nm and smaller designs.

Standard cell libraries: Our design uses the 90nm high-performance GT standard cell libraries from Taiwan Semiconductor Manufacturing Company (TSMC) [10]. The library contains three implementations of cells, each with a different threshold voltage, for balancing power and speed. We allow DesignCompiler and FirstEncounter to pick the appropriate cell implementation for each path.

The memory in our design is a mixture of SRAM memories generated from a commercial memory compiler (used for the large memory structures, such as data caches) and Synopsys DesignWare IP memory building blocks (used for smaller memory structures).

Timing data: Architects prefer to evaluate clock cycle time in a process-independent metric, fanout-of-four (FO4). A design's cycle time in FO4 does not change (much) as the fabrication process changes, thus enabling a more direct comparison of designs across process technologies.

Synthesis tools, however, report delay in absolute terms (nanoseconds). To convert nanoseconds to FO4, we followed academic precedent [11] and used the technique suggested in [12] to measure the absolute delay of one FO4. We synthesized a ring oscillator using the same design flow and top-speed standard cells (LVT) used in our design and measured FO1 (13.8ps). We then multiplied this delay by three to yield an approximation of one FO4 (41.4ps). All timing data presented here is reported in FO4 based upon this measurement.

2.2 Cycle-level functional simulation

To complement our RTL model, we built a corresponding cycle-level simulator of the microarchitecture. The simulator models each major subsystem of the WaveScalar processor (execution, memory, and network) and is used to explore their design in more detail. It also answers basic questions, such as how the sizing of microarchitectural features affect performance. To drive the simulations, we executed the suite of applications described below. These applications were compiled with the DEC Alpha CC compiler and then binary translated into WaveScalar assembly. The assembly files are then compiled with our WaveScalar assembler, and these executables are used by our simulator.

Applications: We used three groups of workloads to evaluate the WaveScalar processor; each focuses on a different aspect of WaveScalar performance. To measure single-threaded performance, we chose a selection of the Spec2000 [13] benchmark suite (*ammp*, *art*, *equake*, *gzip*, *twolf* and *mcf*). To evaluate the processor's media processing performance we use *rawdaudio*, *mgeg2encode*, and *djpeg* from Mediabench [14]. Finally, we use six of the Splash2 benchmarks, *fft*, *lu-continuous*, *ocean-noncontinuous*, *raytrace*, *water-spatial*, and *radix*, to explore multi-threaded performance. We chose these subsets of the three suites because they represent a variety of workloads and our binary translator-based tool-chain can handle them.

3 A WaveScalar Overview

This section provides a brief overview of the WaveScalar architecture and implementation to provide context for the area model and performance results presented in Section 4. More detail about the instruction set is available in [5, 15]. Once we have set the high-level stage in the next subsection, the following three subsections present the portions of the execution, communication, and memory systems that contribute the most to WaveScalar's area budget. This section describes the purpose and function of each component. The appendix supplements this section with a step-by-step example of PE operation.

3.1 The WaveScalar Architecture

WaveScalar is a tagged-token, dynamic dataflow architecture. Like all dataflow architectures (e.g. [16, 17, 18, 19,

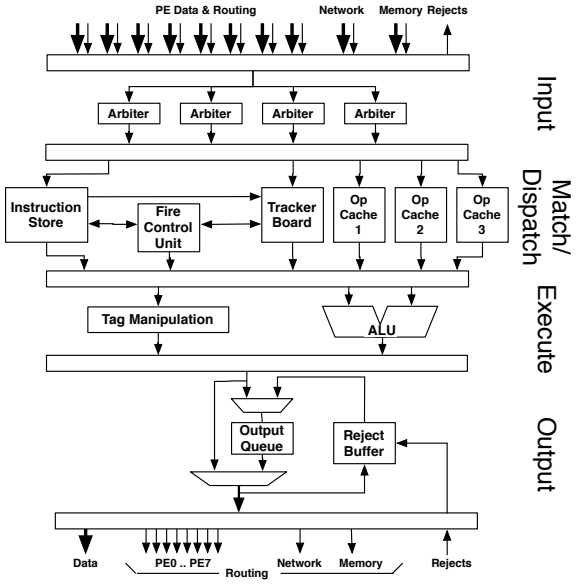

Figure 1. PE Block Diagram: The processing element's structure by pipeline stage. Note that the block at the end of Output is the same as the block at the start of Input since wire delay is spread between the two stages.

20, 21, 22]), its application binary is a program's dataflow graph. Each node in the graph is a single instruction which computes a value and sends it to the instructions that consume it. An instruction executes after all its input operand values arrive according to a principle known as the *dataflow firing rule* [16, 17]. WaveScalar can execute programs written with conventional von Neumann-style memory semantics (i.e. those composed in languages like C/C++) and correctly orders memory operations with a technique called *wave-ordered memory* [5].

PEs form WaveScalar's execution core. From the point of view of the programmer, the WaveScalar execution model provides a PE for each static instruction in an application binary[1]. Since this is clearly not practical (or efficient), the processor contains a smaller pool of PEs and dynamically binds instructions to PEs as an application executes, swapping them in and out on demand. This binding process is *not* the same as instruction fetch on conventional processors. The key difference is that once an instruction is bound to a physical PE, it can remain there for many dynamic executions.

Good instruction placement is critical for good performance. In this work, we use a highly tuned placement algorithm that uses depth-first traversal of the dataflow graph to build chains of dependent instructions that execute sequentially at one processing element. It then assigns those chains to PEs on demand as the program executes. Other work [7] describes the algorithm in detail. Figure 2 illustrates how a program can be mapped into a WaveScalar processor for execution.

A PE contains all the logic for dataflow execution. It has an input interface that receives tokens containing a data value and information that associates the value with a particular instruction. These tokens are stored in a matching table, which is implemented as a small, non-associative cache. Once all tokens for an instruction have arrived, the instruction can be scheduled for execution. An ALU executes the instructions and sends the results to an output network interface, which in turn conveys them to consumer PEs or to the wave-ordered store-buffer (described below). The microarchitecture of the PE will be described in detail in the next section. An evaluation of the best PE configuration (in relation to the rest of the design) is in Section 4.

The wave-ordered store buffer manages load and store requests for PEs. Its key feature is the ability to order these operations in program order, which is critical for correct execution of imperative languages. A description of the microarchitecture of the wave-ordered store buffer is contained in Section 3.3.

The instruction storage system is distributed across the PEs. The data storage system consists of a two-level cache hierarchy backed by main memory. The first level of the hierarchy is a collection of L1 caches that are distributed across the WaveScalar processor. These caches are kept coherent using a MESI-like directory protocol. A banked L2 cache sits between the L1 caches and main memory. The L2 is distributed on the device but banked by address, so no coherence protocol is required [2].

To reduce communication costs, all of these components (PEs, store-buffers, and data-caches) are connected using a hierarchical interconnection structure, depicted in Figure 3. Pairs of PEs are first coupled into *pods* which share ALU results via a common bypass network. Pods are further grouped into *domains*; within a domain, PEs communicate over a set of pipelined busses. Four domains form a *cluster*, which also contains wave-ordered memory hardware (in the store buffer), a network switch, and an L1 data cache. A single cluster, combined with an L2 cache and traditional main memory, is sufficient to run any WaveScalar program, but performance might be poor if the working set of instructions is larger than the cluster's instruction capacity. To build larger, higher performing machines, multiple clusters are connected by a grid-based on-chip network.

3.2 Processing Elements

A set of processing elements (PEs) form the execution resources of a WaveScalar processor. We begin by describing the PE's function and presenting a broad overview of its pipeline stages. We then describe the structures and pipeline stages that have the greatest impact on the PE's area and performance. The appendix contains a detailed operational example of the PE microarchitecture.

The PE (Figure 1) is the heart of a WaveScalar machine. It executes instructions and communicates results over a network. Our RTL implementation uses a PE with five pipeline stages; they are:

1. INPUT: Operand messages arrive at the PE either from itself or another PE. The PE may reject messages if too many arrive in one cycle; the senders will retry on a later cycle.

2. MATCH: Operands enter the *matching table*. The matching table contains a tracker board and operand caches. It determines which instructions are ready to fire and issues eligible instructions by placing their matching table index into the instruction scheduling queue.

3. DISPATCH: The PE selects an instruction from the scheduling queue, reads its operands from the matching table, and forwards them to EXECUTE. If the

[1] More precisely, for each instruction in each programmer-created thread.

[2] This is only true for single-chip WaveScalar systems, of course.

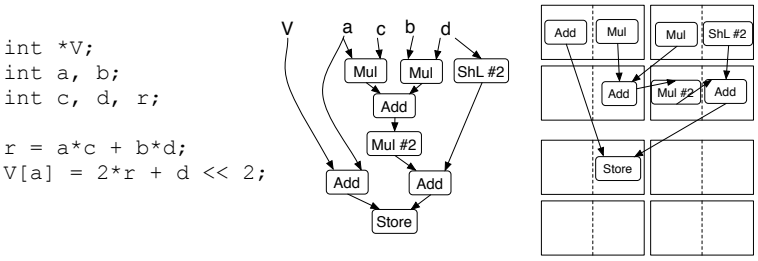

Figure 2. Three views of code in WaveScalar: At left is the C code for a simple computation. Its WaveScalar dataflow graph is shown at center and then mapped onto 2 8-PE domains in a WaveScalar processor at right.

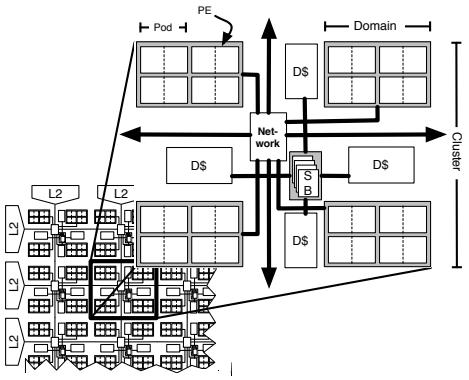

Figure 3. The WaveScalar processor and cluster: The hierarchical organization of the WaveScalar microarchitecture.

destination of the dispatched instruction is local, this stage speculatively issues the consumer instruction to the scheduling queue.

4. EXECUTE: Executes an instruction. Its result goes to the output queue and/or the local bypass network.

5. OUTPUT: An instruction output is sent to its consumer instructions via the intra-domain network. Consumers may be at this PE or at a remote PE.

An instruction store holds the decoded instructions that reside at a PE. To keep it single-ported, the RTL design divides it into several small SRAMs, each holding decoded information needed at a particular stage of the pipeline. The instruction store comprises about 33% of the PE's area (Table 2).

The matching table handles instruction input matching. Implementing this operation cost-effectively is essential to an efficient dataflow machine. The key challenge in designing WaveScalar's matching table is emulating a potentially infinite table with a much smaller physical structure. This problem arises because WaveScalar is a dynamic dataflow architecture e.g. [22, 23, 24, 19] with no limit on the number of dynamic instances of a static instruction with unconsumed inputs. We use a common dataflow technique [19, 18] to address this challenge: the matching table is a specialized cache for a larger, in-memory matching table. New tokens are stored in the matching cache. If a token resides there for a sufficiently long time, another token may arrive that hashes to the same location. In this case, the older token is sent to the matching table in memory.

The matching table is separated into three columns, one for each potential instruction input (certain WaveScalar instructions, such as data steering instructions, can have three inputs[3]). Each column is divided into four banks to allow up to four messages to arrive each cycle. Reducing the number of banks to two reduced performance by 5% on average and 15% for *ammp*. Increasing the number of banks to eight had negligible effect. In addition to the three columns, the matching table contains a *tracker board*, which holds operand tags (wave number and consumer instruction number) and tracks which operands are present in each row of the matching table.

Since the matching table is a cache, we can apply traditional cache optimizations to reduce its miss rate. Our simulations show that 2-way set associativity increases performance by 10% on average and reduces matching table misses (situations when no row is available for an incoming operand) by 41%. 4-way associativity provides less than 1% additional performance, hence the matching table is 2-way. The matching table comprises about 60% of PE area.

To achieve good performance, PEs must be able to execute dependent instructions on consecutive cycles. When DISPATCH issues an instruction with a local consumer of its result, it speculatively schedules the consumer instruction to execute on the next cycle. The schedule is speculative, because DISPATCH cannot be certain that the dependent instruction's other inputs are available. If they are not, the speculatively scheduled consumer is ignored.

To allow back-to-back execution of instructions in different PEs, we combine two PEs into a single pod. PEs in a pod snoop each others bypass networks, but all other parts of their design remain partitioned – separate matching tables, instruction store, etc. Our simulations show that the 2-PE pod design is 15% faster on average than isolated PEs. Increasing the number of PEs in each pod further increases performance but adversely affects cycle time.

3.3 The Memory Subsystem

The WaveScalar processor's memory system has two parts: the wave-ordered store buffers that provide von Neumann memory ordering and a conventional memory hierarchy with distributed L1 and L2 caches. Both components contribute significantly to chip area.

3.3.1 Wave-ordered Interface

WaveScalar provides a memory interface called wave-ordered memory (described in [5]) that enables it to execute programs written in imperative languages (such as C, C++, or Java [6, 25]), by providing the well-ordered memory semantics these languages require.

The store buffers, one per cluster, are responsible for implementing the wave-ordered memory interface that guarantees correct memory ordering. To access memory, processing

[3]The third column is special and supports only single-bit operands. This is because three input instructions in WaveScalar always have one argument which need only be a single bit. Other columns hold full 64 bit operands.

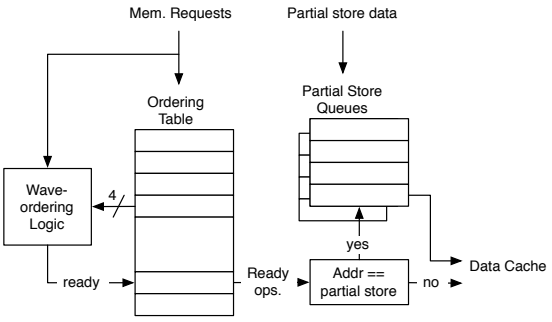

Figure 4. The store buffer: architecture. The main area consumer is the ordering table. The processing logic is pipelined (3 stages, not shown) and of negligible area. Two partial store queues (right) were found to be sufficient for performance.

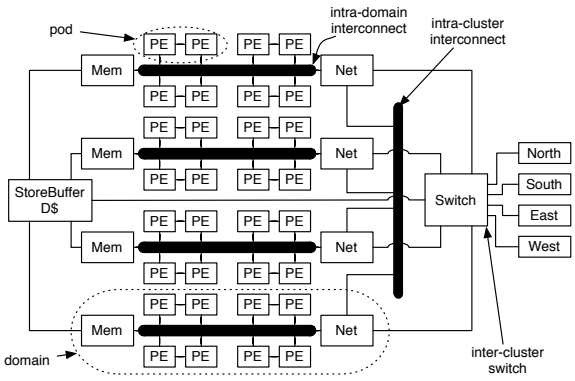

Figure 5. The cluster interconnects: A high-level picture of the interconnects within a cluster.

elements send requests to their local store buffer via a specialized PE in their domain, the MEM pseudo-PE (described below). The store buffer will either process the request or direct it to another buffer via the inter-cluster interconnect. All memory requests for a section of code (called a *wave* in [5]), such as a loop iteration, including requests from both local and remote processing elements, are managed by the same store buffer. Each store buffer can handle four iterations simultaneously.

The store buffer uses a technique called *store decoupling* to process store address and store data messages separately. If a store address arrives and is ready to issue to the data cache before its data value has arrived, the store buffer assigns the store address to a *partial store queue*, where it awaits its data value. In the meantime, any requests that issue from the store buffer and target an address assigned to a partial store queue are placed in that partial store queue as well. When the missing data value finally arrives, the partial store queue can issue all its requests in quick succession.

Our design includes two partial store queues, each of which can hold four memory requests. Each partial store queue has one read and one write port. In addition, a 2-entry associative table detects whether an issued memory operation should be written to a partial store queue or be sent to the cache. Adding partial store queues increases performance between 5 and 20%, depending upon the application. Adding more than two partial store queues provides a negligible additional increase in performance, but makes achieving a short cycle time difficult.

All the store buffer hardware for one cluster, including the partial store queues, occupies 2.6mm^2 in 90nm technology or approximately 6.2% of the cluster, depending on the cluster's configuration. The size of the store buffer is an architectural parameter exposed to the ISA, so its area requirements are fixed. Each store buffer can handle four wave-ordered memory sequences at once.

3.3.2 Data-cache System

WaveScalar provides two levels of data cache. Each cluster contains an L1 cache, and the banks of an L2 cache and a coherence directory surround the array of clusters. The L1 data cache is 4-way set associative and has 128-byte lines. An L1 hit costs 3-cycles (2 cycles SRAM access, 1 cycle processing), which can overlap with store buffer processing. A directory-based, MESI coherence protocol keeps the L1 caches coherent [26]. All coherence traffic travels over the inter-cluster interconnect. The L2's hit delay is 20-30 cycles, depending upon address and distance to a requesting cluster. Main memory latency is modeled at 200 cycles.

An area analysis of the data-cache system appears with the rest of the results in Section 4.

3.4 Interconnection Network Scalability

Section 3.2 describes WaveScalar's execution resource, the PE. PEs communicate by sending and receiving data via a hierarchical, on-chip interconnect. This hierarchy has four levels: intra-pod, intra-domain, intra-cluster and inter-cluster. Figure 5, which depicts a single cluster, illustrates all on-chip networks. While these networks serve the same purpose – transmission of instruction operands and memory values – they have significantly different implementations. Section 3.2 described the intra-pod interconnect. Here, we present the salient features of the remaining three.

3.4.1 The intra-domain interconnect

The intra-domain interconnect is broadcast-based. Each of a domain's PEs has a dedicated result bus that carries a single data result to the other PEs in its domain. Widening the broadcast busses makes little sense, since the PE can only generate one output value per cycle (it has only one ALU) and fewer than 1% of messages need to be broadcast more than once (e.g. when a receiver cannot yet handle an incoming message).

In addition to the normal PEs, each domain contains two pseudo-PEs (called MEM and NET) that serve as gateways to the memory system and PEs in other domains or clusters, respectively. PEs and pseudo-PEs communicate over the intra-domain interconnect using identical interfaces.

Although the intra-domain interconnect is the largest broadcast structure in the processor, it has far less impact on total chip area than its size might indicate, because the interconnect wires reside mostly in metal layers above the PE

WaveScalar Capacity	4K static instructions (128 per PE)		
PEs per Domain	8 (4 pods)	Domains / Cluster	4
PE Input Queue	128 entries, 4 banks	Network Latency	within Pod: 1 cycle within Domain: 5 cycles within Cluster: 9 cycles inter-Cluster: 9 + cluster dist.
PE Output Queue	4 entries, 2 ports (1r, 1w)		
PE Pipeline Depth	5 stages		
L1 Cache	32KB, 4-way set associative, 128B line, 4 accesses per cycle	Network Switch	2-port, bidirectional
Main RAM	200 cycle latency		

Table 1. Microarchitectural parameters: The configuration of the baseline WaveScalar processor

logic. Accessing the interconnect and accommodating vias causes the PE logic to expand by 7%. This expansion is essentially the intra-domain network area.

3.4.2 The intra-cluster interconnect

The intra-cluster interconnect is a small network that shuttles operands between the domains. The NET pseudo-PEs of the four domains within a cluster are connected to each other and to the inter-cluster network switch in a complete, point-to-point network. Every point-to-point link is capable of moving one operand per cycle in each direction, although the NET pseudo-PEs can only introduce a single operand to their respective domains each cycle. Area for the intra-cluster interconnect is negligible. Its largest resource, the wires, reside in metal layers above the intra-domain network.

3.4.3 The inter-cluster interconnect

The inter-cluster interconnect is responsible for all long-distance communication in the WaveScalar processor, including operands traveling between PEs in different clusters and coherence traffic for the L1 data caches. At each cluster, the network switch routes messages between six input/output ports. Four of the ports lead to the network switches in the four cardinal directions, one is shared among the domains' NET pseudo-PEs, and one is dedicated to the store buffer and L1 data cache.

Each input/output port supports the transmission of up to two operands per cycle. The inter-cluster network provides two virtual channels which the interconnect uses to prevent deadlock [27]. Each output port contains two 8-entry output queues, one for each virtual network. The output queues rarely fill completely (less than 1% of cycles). Our experiments found that lowering bandwidth to one operand per cycle significantly hurt performance (by 52% on average), while increasing it to four had a negligible effect. In this paper, we do not examine variations of the switch configuration further, primarily because it comprises an extremely small area compared to the rest of the design. Its queues consume negligible area, and the inter-cluster wires comprising the switch fabric are routed through the upper layers of metal on the chip. All told, the inter-cluster interconnect represents only 1% of the total chip area for *all* of the configurations we consider in this work.

4 Evaluation

The architecture described in the previous sections defines a large set of WaveScalar processors of varying sizes and configurations. At one end of the spectrum is a small WaveScalar processor, comprising just a single cluster, which would be suitable for small, single-threaded or embedded applications. At the other end, a supercomputer processor might contain tens of clusters and hundreds or thousands of processing elements. The ability to move easily along this design continuum is a key objective of tiled architectures. A second objective is that they be able to tolerate such drastic changes in area by localizing data communication, thereby reducing latency. This section explores how well the WaveScalar architecture achieves these goals.

We begin in Section 4.1 with a detailed look at the area budget of a particular WaveScalar processor configuration. Then, to understand a larger design space, Section 4.2 uses data from our RTL synthesis to develop an area model that describes the area requirements for a range of designs. We use the resulting model to enumerate a large class of WaveScalar processor designs that could be built in modern process technology. We evaluate these designs using a suite of single- and multi-threaded workloads and use the results to perform a area/performance pareto analysis of the WaveScalar design space covered by our RTL design and area model. Lastly, in Section 4.3, we examine changes in the network traffic patterns as the size of the WaveScalar processor increases.

4.1 Area and timing results

Our RTL toolchain provides both area and delay values for each component of the WaveScalar processor. For this study, we restrict ourselves to processors that achieve a clock rate of 22 FO4, which occurs for a wide range of designs in our Verilog model. This is the shortest cycle time allowed by the critical path within the PE. For most configurations, the critical path is through the execution unit's integer multiplier, when using operands from the other PE in the pod. However, enlarging the matching cache or instruction cache memory structures beyond 256 entries makes paths in the MATCH and DISPATCH stages critical paths. Floating point units are pipelined to avoid putting floating-point execution on the critical path. INPUT and OUTPUT devote 9 and 5 FO4 respectively to traversing the intra-domain network, so there is no need for an additional stage for intra-domain wire delay.

Since the critical path is the ALU for designs with smaller than 256-entry matching caches and 256-entry instruction caches, we can resize these structures downward for optimum area-performance without dramatically altering the cycle time. This allows us to evaluate a large number of potential processing element designs based on area without worrying about an accompanying change in cycle time. We confirmed this by synthesizing designs with 16- to 256-entry matching caches and with 8- to 256-entry instruction caches. The clock cycle for these configurations changed by less than 5% until the structures reached 256 entries, at which point the cycle time increased by about 21% for the matching cache and 7% for the instruction cache. These latencies and structure size limits for our study are summarized in Table 1 and

	Area in PE	Area in Domain	Area in Cluster	% of PE	% of Domain	% of Cluster
PE						
INPUT	0.01mm^2	0.09mm^2	0.37mm^2	1.2%	1.1%	0.9%
MATCH	0.58mm^2	4.60mm^2	18.41mm^2	61.0%	55.2%	43.3%
DISPATCH	0.01mm^2	0.05mm^2	0.18mm^2	0.6%	0.6%	0.4%
EXECUTE	0.02mm^2	0.19mm^2	0.77mm^2	2.5%	2.3%	1.8%
OUTPUT	0.02mm^2	0.14mm^2	0.55mm^2	1.8%	1.7%	1.3%
instruction store	0.31mm^2	2.47mm^2	9.88mm^2	32.8%	29.6%	23.2%
total	0.94mm^2	7.54mm^2	30.16mm^2	100%	90.5%	71.0%
Domain						
MemPE		0.13mm^2	0.53mm^2		1.6%	1.2%
NetPE		0.13mm^2	0.53mm^2		1.6%	1.2%
8×PE		7.54mm^2	30.16mm^2		90.5%	71.0%
FPU		0.53mm^2	2.11mm^2		6.3%	5.0%
total		8.33mm^2	33.32mm^2		100%	78.4%
Cluster						
4× domain			33.32mm^2			78.4%
network switch			0.37mm^2			0.9%
store buffer			2.62mm^2			6.2%
data cache			6.18mm^2			14.5%
total			42.50mm^2			100.0%

Table 2. A cluster's area budget: A breakdown of the area required for a single cluster.

Table 3, respectively.

Table 2 shows how the die area is spent for the baseline design described in Table 1. Note that the vast majority of the cluster area (71%) is devoted to PEs. Also, as shown in the table, almost all of the area, ∼80%, is spent on SRAM cells which make up the instruction stores, matching caches, and L1 data caches.

4.2 Area model and Pareto analysis

Many parameters affect the area required for WaveScalar designs. Our area model considers the seven parameters with the strongest effect on the area requirements. Table 3 summarizes these parameters (top half of the table) and how they combine with data from the RTL model to form the total area, WC_{area} (bottom of the table). For both the matching table and instruction store, we synthesized versions from 8 to 128 entries to confirm that area varies linearly with capacity. For the L1 and L2 caches, we used the area of 1KB and 1MB arrays provided by a memory compiler to perform a similar verification. The "Utilization factor" is a measure of how densely the tools managed to pack cells together, while still having space for routing. Multiplying by its inverse accounts for the wiring costs in the entire design.

The area model ignores some minor effects. For instance, it assumes that wiring costs do not decrease with fewer than four domains in a cluster, thereby overestimating this cost for small clusters. Nevertheless, the structures accounting for most of the silicon area (80% as discussed in Section 4.1) are almost exactly represented.

The area model contains several parameters that enumerate the range of possible WaveScalar processor designs. For parameters D (domains per cluster), P (processors per domain), V (instructions per PE) and M (matching-table entries), we set the range to match constraints imposed by the RTL model. As mentioned in Section 4.1, increasing any of these parameters past the maximum value impacts cycle time. The minimum values for M and V reflect restrictions on minimum memory array sizes in our synthesis toolchain.

The ranges in the table allow for over twenty-one thousand WaveScalar processor configurations, but many of them are clearly poor, unbalanced designs, while others are extremely large (up to $12{,}000\text{mm}^2$). We can reduce the number of designs dramatically if we apply some simple rules.

First, we bound the die size at 400mm^2 in 90nm to allow for aggressively large yet feasible designs. Next, we remove designs that are clearly inefficient. For instance, it makes no sense to have more than one domain if the design contains fewer than eight PEs per domain. In this case, it is always better to combine the PEs into a single domain, since reducing the domain size does not reduce the cycle time (which is set by the PE's EXECUTE pipeline stage) but does increase communication latency. Similarly, if there are fewer than four domains in the design, there should be only one cluster. Applying these rules and a few more like them reduces the number of designs to 201.

We evaluated all 201 design points on our benchmark set. For the Splash applications, we ran each application with a range of thread counts on each design and report results for the best-performing thread count. Figure 6 shows the results for each group of applications. For all our performance results we report AIPC instead of IPC. AIPC is the number of Alpha-equivalent instructions executed per cycle. All additional WaveScalar-specific instructions [28, 15] are not included, so that performance is more intuitive. Each point in the graph represents a configuration, and the circled points are pareto optimal configurations (i.e., there are no configurations that are smaller *and* achieve better performance). We discuss multi- and single-threaded applications separately and then compare WaveScalar to two other architectures in terms of area efficiency.

Splash2 Figure 6 shows that Splash2's use of multiple threads allows it to take advantage of additional area very effectively, as both performance and area increase at nearly the same rate. The clusters of pareto optimal points are reflected in Table 4, which contains the optimal configurations for Splash2 divided into five groups of designs with similar performance (column "Avg. AIPC").

Parameter	Symbol	Description	Range
Clusters	C	Number of clusters in the WaveScalar processor	$1 \ldots 64$
Domains/cluster	D	Number of domains per cluster	$1 \ldots 4$
PEs/domain	P	Number of PEs per domain	$2 \ldots 8$
PE virtualization degree	V	Instructions capacity of each PE	$8 \ldots 256$
Matching table entries	M	Number of matching table entries	$16 \ldots 128$
L1 Cache size	$L1$	KB of L1 cache/cluster	$8 \ldots 32$
L2 Cache size	$L2$	total MB of L2 cache	$0 \ldots 32$
Area component	**Symbol**	**Value**	
PE matching table	M_{area}	$= 0.004\text{mm}^2/\text{entry}$	
PE instruction store	V_{area}	$= 0.002\text{mm}^2/\text{instruction}$	
Other PE components	e_{area}	$= 0.05\text{mm}^2$	
Total PE	PE_{area}	$= M \times M_{\text{area}} + V \times V_{\text{area}} + e_{\text{area}}$	
Pseudo-PE	PPE_{area}	$= 0.1236\text{mm}^2$	
Domain	D_{area}	$= 2 \times PPE_{\text{area}} + P \times PE_{\text{area}}$	
Store buffer	SB_{area}	$= 2.464\text{mm}^2$	
L1 cache	$L1_{\text{area}}$	$= 0.363\text{mm}^2/\text{KB}$	
Network switch	N_{area}	$= 0.349\text{mm}^2$	
Cluster	C_{area}	$= D \times D_{\text{area}} + SB_{\text{area}} + L1 \times L1_{\text{area}} + N_{\text{area}}$	
L2 area	$L2_{\text{area}}$	$= 11.78\text{mm}^2/\text{MB}$	
Utilization factor	U	$= 0.94$	
Total WaveScalar processor area	WC_{area}	$= \frac{1}{U}(C \times C_{\text{area}}) + L2_{\text{area}}$	

Table 3. WaveScalar processor area model.

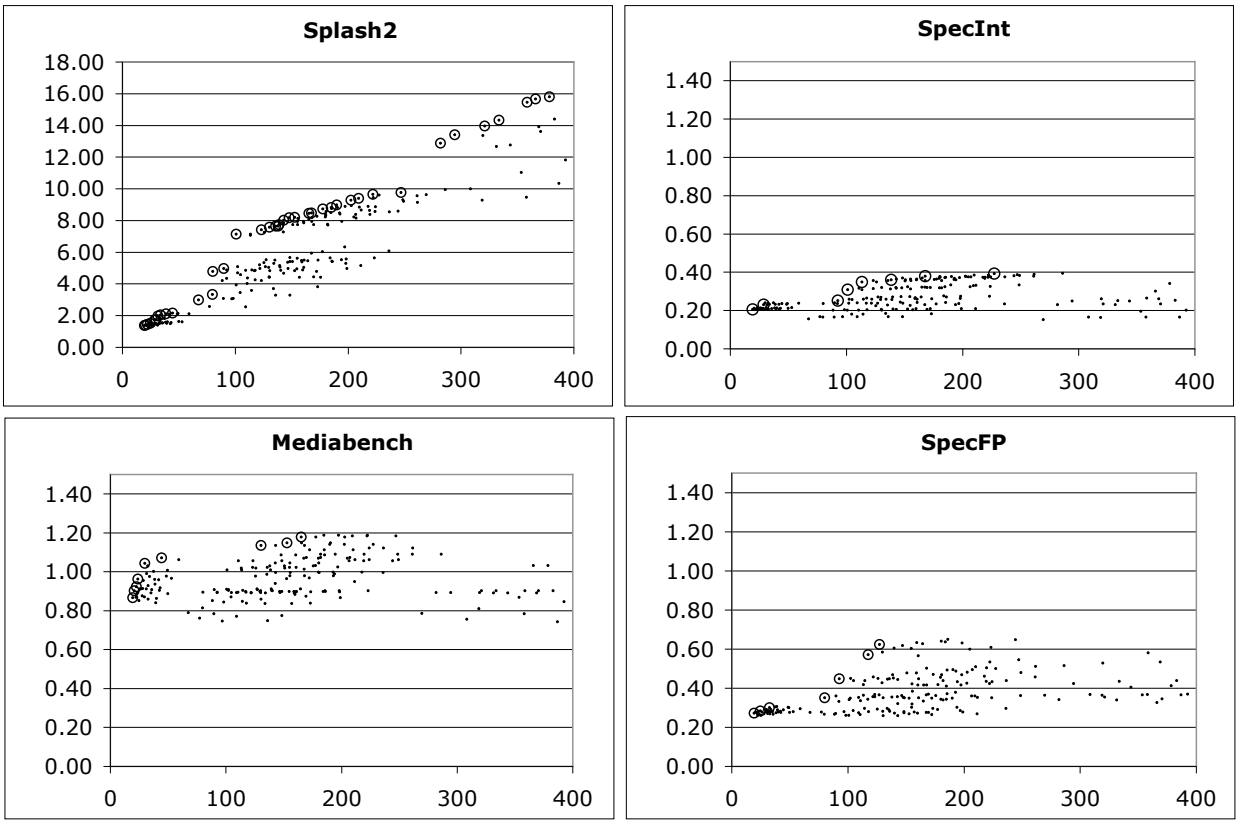

Figure 6. Pareto-optimal WaveScalar designs: Small points are all designs. Circles are pareto-optimal points. Vertical axis measures AIPC. Horizontal axis is mm^2. Note the difference in AIPC scale for the Splash2 data.

Group A is the single-cluster configurations. Designs 1, 2, 5, and 6 contain only two domains per cluster. Designs 5 and 6 outperform smaller, four-domain designs (3 and 4) because pressure in the L1 data cache outweighs the benefits of additional processing elements.

Moving to four clusters without an L2 cache (Group B) increases the chip's size by 52% and its performance by 38%. Doubling the size of the L1 provides an additional 11% performance boost.

Group C (configurations 13 and 14) "trades-in" large L1 caches for an on-chip L2. For configuration 13, this results in a 43.5% improvement in performance for a negligible increase in area.

The first four 4-cluster designs (Groups B and C) all have much smaller virtualization degrees than the best-performing single-cluster designs (64 vs. 128-256). Configuration 15, the first in Group D, has 128 instructions per PE. The change leads to 43% more performance and requires only 12% additional area.

The remaining designs in Group D increase slowly in area and performance. Virtualization degree is most important to performance followed by matching table size and cache capacity. With few exceptions, the gains in performance are smaller than the increase in area that they require. As a result, the area efficiency of the designs falls from 0.07 AIPC/mm^2 to 0.04 AIPC/mm^2.

The configurations in Group E (33-39) have 16 clusters. The transition from 4 to 16 clusters mimics the transition from 1 to 4: The smallest 16 cluster designs have less virtualization and increase their performance by increasing matching table capacity and cache sizes. The larger designs "trade-in" matching table capacity for increased virtualization.

The results demonstrate that WaveScalar is scalable both in terms of area efficiency and peak performance. Design 3 is the most area efficient design, and design 15 uses the same cluster configuration to achieve the same level of efficiency. As a result, the designs are approximately a factor of 5 apart in both area and performance. Scaling the same cluster configuration to a 16-cluster machine (configuration 38) reduces the area efficiency to 0.04 AIPC/mm^2, however. A more efficient alternative (0.05 AIPCmm^2) is configuration 33, which corresponds to scaling up configuration 13 and using a slightly smaller L2 data cache.

Design 10 outperforms all the other 1 cluster designs (2.2 AIPC), and configuration 28 uses the same cluster design to achieve 4.2× the performance (9 AIPC) in 4.6× the area. This level of performance is within 5% of the best performing 4 cluster design (configuration 32), demonstrating scalability of raw performance.

This slightly imperfect scalability, however, (the last 5%) raises a complex question for chip designers. For instance, if an implementation of configuration 38's cluster is available (e.g., from an implementation of configuration 3), is it more economic to quickly build that larger, slightly less efficient design or to expend the design and verification effort to implement configuration 33 instead? At the very least, it suggests tiled architectures are not inherently scalable, but rather scalability claims must be carefully analyzed.

Single-threaded workloads The data for all three groups of single-threaded workloads follow the same trend (see Figure 6). The configurations fall into three clumps depending on how many clusters they utilize. Both SpecINT and SpecFP see the most benefit from four cluster designs. On a single cluster they achieve only 58% and 46% (respectively) of the peak performance they attain on four clusters. Mediabench sees smaller gains (9%) from four clusters.

None of the workloads productively utilize a 16 cluster design. In fact, performance decreases for these configurations because instructions become more spread out, increasing communication costs. This agrees with prior WaveScalar research [5].

Area efficiency and other architectures The above results show that WaveScalar's area efficiency scales as we build larger WaveScalar processors, but they do not show whether WaveScalar is efficient compared to other architectures. To investigate this we compare WaveScalar's area efficiency to two other architectures: the Alpha 21264 and the Sun's Niagara processor.

To compare to the Alpha we use `sim-alpha` configured to model the Alpha EV7 [29, 30], but with the same L1, L2, and main memory latencies we model for WaveScalar. To estimate the size of the Alpha, we examined a die photo of the EV7 in 180nm technology [30, 31]. The entire die is 396mm^2. From this, we subtracted the area devoted to several components that our area model does not include (e.g., IO pads, and inter-chip network controller). We estimate the remaining area to be $\sim$291mm^2. Scaling to 90nm technology yields $\sim$72mm^2.

For our single threaded workloads the Alpha achieves an area efficiency of 0.008 AIPC/mm^2 (0.6 AIPC). Depending on the configuration, WaveScalar's efficiency for the single-threaded applications is between 0.004 AIPC/mm^2 (0.71 AIPC; configuration 31) and 0.033 AIPC/mm^2 (0.45 AIPC; configuration 1). WaveScalar configuration 4 closely matches the Alpha's performance with an efficiency of 0.024 AIPC/mm^2, three times that of the Alpha.

The 21264 is a single-threaded processor, but WaveScalar can execute multiple threads as well. To gauge its efficiency in this arena, we compare to Sun's Niagara processor [32]. Niagara is an 8-way CMP targeted at large multi-threaded workloads. We are unaware of any published evaluation of Splash2 on Niagara, but [32] shows Niagara running at 4.4 IPC (0.55 IPC/core) at 1.2 GHz (an approximately 10% faster clock than WaveScalar's). Niagara's die measures 379mm^2, giving an efficiency of 0.01 IPC/mm^2. Since each Niagara core is single-issue and in-order, and all eight cores share a floating point unit, the theoretical maximum efficiency is 0.02 IPC/mm^2. The least efficient pareto optimal WaveScalar design (configuration 39) is twice as efficient (0.04 AIPC/mm^2).

4.3 Network traffic

One goal of WaveScalar's hierarchical interconnect is to isolate as much traffic as possible in the lower levels of the hierarchy, namely, within a PE, a pod or a domain. Figure 7 breaks down all network traffic according to these levels. It reveals the extent to which the heirarchy succeeds on all three workloads, and for the parallel applications, on a variety of WaveScalar processor sizes. On average 40% of network traffic travels from a PE to itself or to the other PE in its pod, and 52% of traffic remains within a domain. For multi-cluster configurations, on average just 1.5% of traffic traverses the inter-cluster interconnect. The graph also distinguishes between operand data and memory/coherence traffic. Operand data accounts for the vast majority of messages, 80% on average, with memory traffic less than 20%.

These results demonstrate the scalability of communication performance on the WaveScalar processor. Applications that require only a small patch of the processor, such as Spec, can execute without ever paying the price for long distance communication. In addition, the distribution of traffic types barely changes with the number of clusters, indicating that the interconnect partitioning scheme is scalable. Message

Group	Id	Clusters	Domains/ Cluster	PEs/ Domain	Virt.	Matching Entries	$L1$ (KB)	$L2$ (MB)	Inst. Cap.	Area (mm^2)	Avg. AIPC	AIPC/ mm^2	Area Increase	AIPC Increase
A	1	1	2	8	256	16	8	0	4096	19	1.4	0.07	na	na
	2	1	2	8	256	32	8	0	4096	21	1.4	0.07	6.3%	0.7%
	3	1	4	8	128	16	8	0	4096	22	1.5	0.07	7.4%	4.0%
	4	1	4	8	128	32	8	0	4096	25	1.5	0.06	11.1%	5.0%
	5	1	2	8	256	16	32	0	4096	29	1.7	0.06	16.6%	13.2%
	6	1	2	8	256	32	32	0	4096	30	1.8	0.06	4.3%	1.8%
	7	1	4	8	128	16	32	0	4096	31	2.0	0.06	5.1%	11.9%
	8	1	4	8	128	32	32	0	4096	34	2.0	0.06	7.8%	3.7%
	9	1	4	8	128	64	32	0	4096	39	2.1	0.05	14.5%	3.2%
	10	1	4	8	256	32	32	0	8192	44	2.2	0.05	14.5%	3.1%
B	11	4	4	8	64	16	8	0	8192	67	3.0	0.04	51.9%	37.8%
	12	4	4	8	64	16	16	0	8192	80	3.3	0.04	18.4%	11.5%
C	13	4	4	8	64	16	8	1	8192	80	4.8	0.06	0.2%	43.5%
	14	4	4	8	64	32	8	1	8192	90	5.0	0.06	12.3%	4.0%
D	15	4	4	8	128	16	8	1	16384	101	7.2	0.07	12.5%	43.4%
	16	4	4	8	128	32	16	1	16384	123	7.4	0.06	22.0%	3.9%
	17	4	4	8	128	64	8	1	16384	130	7.6	0.06	5.9%	2.0%
	18	4	4	8	128	32	16	2	16384	136	7.6	0.06	4.1%	0.9%
	19	4	4	8	128	16	32	1	16384	138	7.7	0.06	1.8%	0.1%
	20	4	4	8	128	16	8	4	16384	138	7.7	0.06	0.4%	1.0%
	21	4	4	8	128	64	8	2	16384	143	8.0	0.06	3.1%	3.6%
	22	4	4	8	128	32	32	1	16384	148	8.2	0.06	3.5%	2.1%
	23	4	4	8	256	32	8	1	32768	153	8.2	0.05	3.4%	0.3%
	24	4	4	8	256	32	8	2	32768	165	8.5	0.05	8.2%	3.2%
	25	4	4	8	128	64	32	1	16384	167	8.5	0.05	1.3%	0.2%
	26	4	4	8	256	32	16	2	32768	178	8.7	0.05	6.1%	3.0%
	27	4	4	8	256	64	8	2	32768	185	8.8	0.05	4.1%	1.0%
	28	4	4	8	256	32	32	1	32768	190	9.0	0.05	2.7%	2.0%
	29	4	4	8	256	32	32	2	32768	202	9.3	0.05	6.6%	3.2%
	30	4	4	8	256	64	32	1	32768	209	9.4	0.04	3.5%	1.1%
	31	4	4	8	256	64	32	2	32768	222	9.7	0.04	6.0%	2.9%
	32	4	4	8	256	64	32	4	32768	247	9.8	0.04	11.3%	1.2%
E	33	16	4	8	64	16	8	1	32768	282	12.9	0.05	14.1%	31.8%
	34	16	4	8	64	16	8	2	32768	294	13.4	0.05	4.4%	4.0%
	35	16	4	8	64	32	8	1	32768	321	14.0	0.04	9.1%	4.1%
	36	16	4	8	64	32	8	2	32768	334	14.3	0.04	3.9%	2.6%
	37	16	4	8	64	32	8	4	32768	359	15.5	0.04	7.5%	8.0%
	38	16	4	8	128	16	8	1	65536	366	15.7	0.04	2.0%	1.3%
	39	16	4	8	128	16	8	2	65536	378	15.8	0.04	3.4%	0.9%

Table 4. Pareto optimal configurations for Splash2.

latency does increase with the number of clusters (by 12% from 1 to 16 clusters), but as we mentioned above, overall performance still scales linearly. One reason for the scalability is that the WaveScalar instruction placement algorithms isolate individual Splash threads into different portions of the die. Consequently, although the average distance between two clusters in the processor increases from 0 (since there is only one cluster) to 2.8, the average distance that a message travels increases by only 6%. For the same reason, network congestion increases by only 4%.

4.4 Discussion

We have presented a design space analysis for WaveScalar processors implemented in an ASIC tool flow. However, our conclusions also apply to full-custom WaveScalar designs and, in some ways, to other tiled architectures such as CMPs.

The two main conclusions would not change in a full-custom design. For the first conclusion, that WaveScalar is inherently scalable both in terms of area efficiency and raw performance, our data provide a lower bound on WaveScalar's ability to scale. Custom implementation should lead to smaller designs, faster designs, or both. Our second conclusion, that WaveScalar's area efficiency compares favorably to more conventional designs, would hold in a full custom design for the same reason.

WaveScalar's hierarchical interconnect would serve well as the interconnect for an aggressive CMP design, and our results concerning the scalability and performance of the network would apply to that domain as well. In a conventional CMP design, the network carries coherence traffic, but as researchers strive to make CMPs easier to program it might make sense to support other types of data, such as MPI messages, as well. In that case, our data demonstrate the value of localizing communication as much as possible and the feasibility of using a single network for both coherence traffic and data messaging.

5 Related Work

Several research groups have proposed tiled architectures, with widely varying tile designs. Smart Memories [2] provides multiple types of tiles (e.g., processing elements, reconfigurable memory elements). This approach allows greater freedom in configuring an entire processor, since the mix of tiles can vary from one instantiation to the next, perhaps avoiding the difficulties in naive scaling that we found in our study.

TRIPS [3, 4] provides two levels of tiles. The processor consists of a uniform array of functional units that combine to form a processor. The processor itself can be tiled as well. The TRIPS group is building a prototype, but they have not yet published a systematic study of area/performance tradeoffs.

The RAW project [11] uses a simple processor core as a tile and builds a tightly-coupled multiprocessor. One study of the RAW architecture [33] shares similar goals to ours, but it takes a purely analytical approach and creates models for both processor configurations and applications. That study was primarily concerned with finding the optimal configuration for a particular application and problem size.

FPGAs can be viewed as tiled architectures and can offer insight into the difficulties tiled processor designers may face. FPGAs already provide heterogeneous arrays of tiles (e.g., simple lookup tables, multipliers, memories, and even small RISC cores) and vary the mix of tiles depending on the

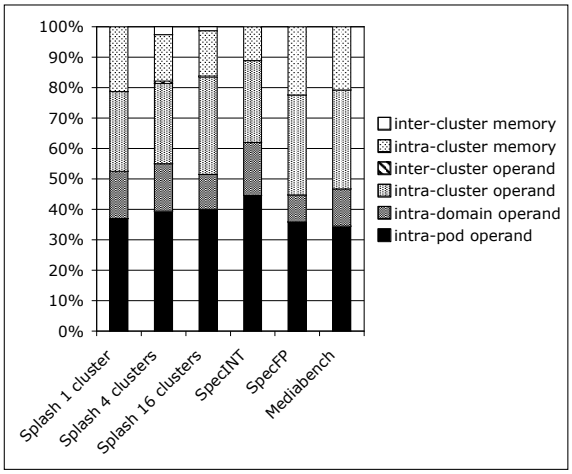

Figure 7. The distribution of traffic in the WaveScalar processor: The vast majority of traffic in the WaveScalar processor is confined within a single cluster and, for many applications, over half travels only over the intra-domain interconnect.

size of the array.

6 Conclusion

This paper explored the area/performance trade-offs in a tiled WaveScalar architecture. We use RTL synthesis and cycle-level simulation to perform a pareto analysis of designs with a 22 FO4 clock cycle ranging from 19mm^2 to 378mm^2. Using applications from SPEC, Splash2, and Mediabench, we investigated over 200 WaveScalar processor configurations.

Exploring the design space reveals some interesting conclusions. First, WaveScalar processors tuned for either area efficiency or maximum performance scale across a wide range of processor sizes. Second, the WaveScalar architecture is more efficient than an aggressive out-of-order superscalar and a modern 8-way CMP at converting silicon area into bottom-line performance.

Finally, we explored the behavior of WaveScalar's hierarchical, on-chip interconnect. Without a scalable interconnect, a tiled architecture cannot scale efficiently. Our interconnect is very effective in this regard. Over 50% of messages in WaveScalar stay within a domain and over 80% stay within a cluster, with memory accounting for nearly all inter-cluster traffic.

7 Appendix

In Section 3 we described the WaveScalar processor architecture. This appendix contains an example illustrating the pipeline operation for a PE. Figure 8 illustrates the two key differences between the PE pipeline and a conventional processor pipeline. Data values, instead of instructions, flow through this pipeline. Also, instructions A and B are speculatively scheduled in order to execute on consecutive cycles. In the sequences shown in Figure 8, A's result is forwarded to B when B is in EXECUTE. In the diagram, $X[n]$ is the nth input to instruction X. Five consecutive cycles are depicted; before the first of these, one input each from instructions A and B have arrived and reside in the matching table. The "clouds" in the dataflow graph represent results of instructions at other processing elements, which have arrived from the input network.

Cycle 0: Operand $A[0]$ arrives and INPUT accepts it.
Cycle 1: MATCH writes $A[0]$ into the matching table and, because both its inputs are now available, places a pointer to A's entry in the matching table in the scheduling queue.
Cycle 2: DISPATCH chooses A for execution, reads its operands and sends them to EXECUTE. At the same time, it recognizes that A's output is destined for B. In preparation for this producer-consumer handoff, a pointer to B's matching table entry is inserted into the speculative fire queue.
Cycle 3: DISPATCH reads $B[0]$ from the matching table and sends it to EXECUTE. EXECUTE computes the result of A, which is $B[1]$.
Cycle 4: EXECUTE computes the result of instruction B using $B[0]$ and the result from the bypass network, $B[1]$.
Cycle 5 (not shown): OUTPUT will send B's output to Z.

Acknowledgments:
This work has been made possible through the generous support of an NSF CAREER Award (ACR-0133188), ITR grant (CCR-0325635), and doctoral fellowship (Swanson); Sloan Research Foundation Award (Oskin); Intel Fellowships (Swanson, Mercaldi); ARCS Fellowships (Putnam, Schwerin); and support from Intel and Dell. We would like to thank our sponsors, the reviewers for their helpful feedback, Seth Bridges for tool-chain support, and the Synopsys and Cadence support staff for their assistance.

References

[1] 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.

[2] 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.

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

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

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

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

[7] M. Mercaldi, "An instruction placement model for distributed ilp architectures," Master's thesis, University of Washington, 2005.

[8] "Synopsys website." http://www.synopsys.com.

[9] "Cadence website." http://www.cadence.com.

[10] "TSMC 90nm technology platform." http://www.tsmc.com/download/english/a05_literature/90nm_Brochure.pdf.

[11] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams," in *Proceedings of the 31st annual international symposium on Computer architecture*, p. 2, IEEE Computer Society, 2004.

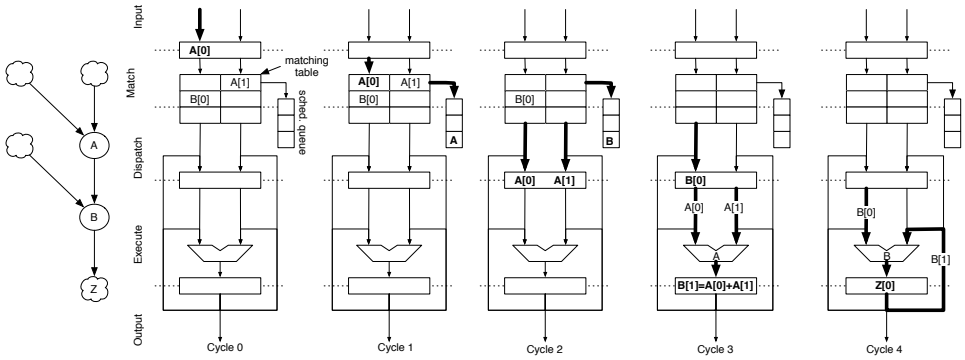

Figure 8. The flow of operands through the PE pipeline and forwarding networks: The figure is described in detail in the text.

[12] D. Chinnery and K. Keutzer, *Closing the Gap Between ASIC & Custom.* Kluwer Academic Publishers, 2003.

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

[14] 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.

[15] "The wavescalar architecture." In submission to ACM Transactions on Computer Systems, TOCS.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," *SIGARCH Comput. Archit. News*, vol. 16, no. 2, pp. 280–298, 1988.

[27] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Trans. Comput.*, vol. 36, no. 5, pp. 547–553, 1987.

[28] S. Swanson, A. Putnam, K. Michelson, M. Mercaldi, A. Petersen, A. Schwerin, M. Oskin, and S. Eggers, "The microarchitecture of a pipelined wavescalar processor: An rtl-based study," Tech. Rep. TR-2004-11-02, University of Washington, 2005.

[29] R. Desikan, D. Burger, S. Keckler, and T. Austin, "Sim-alpha: a validated, execution-driven alpha 21264 simulator," Tech. Rep. TR-01-23, UT-Austin Computer Sciences, 2001.

[30] A. J. et. al., "A 1.2ghz alpha microprocessor with 44.8gb/s chip pin bandwidth," in *IEEE International Solid-State Circuits Conference*, vol. 1, pp. 240–241, 2001.

[31] K. Krewel, "Alpha ev7 processor: A high-performance tradition continues," *Microprocessor Report*, April 2005.

[32] J. Laudon, "Performance/watt: the new server focus," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 5–13, 2005.

[33] C. A. Moritz, D. Yeung, and A. Agarwal, "Exploring performance-cost optimal designs for raw microprocessors," in *Proceedings of the International IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM98*, April 1998.