The Microarchitecture of a Pipelined WaveScalar Processor: An RTL-based Study

Submitted for blind review

Abstract:

WaveScalar is a recently introduced architecture designed to capitalize on the vast number of transistors available on modern processes. Prior work introduced the architecture and used simulation-based results to demonstrate its performance-efficiency compared to conventional designs. But can it really be built in commercially viable area budgets and will it achieve a clock speed comparable to more conventional superscalars?

This paper answers these questions. We have designed a synthesizable RTL model of the WaveScalar microarchitecture, called the WaveCache. This includes its execution substrate, memory system, and interconnect. Using the TSMC 90nm process and latest design tools, this model synthesizes to a chip of 252mm^2 and achieves a clock rate of 25 FO4. This paper describes its RTL implementation and couples it with results from cycle-level simulation to illustrate the key performance-area-delay trade-offs in WaveCache design.

Keywords: WaveScalar, Dataflow computing, ASIC

1 Introduction

When computer architects develop new designs for improving superscalar processor performance, there is a wealth of prior literature and community knowledge to draw upon to place the potential clock-cycle and area impacts of these designs in context. Researchers that propose radically different microarchitectures, however, have historically turned to building or synthesizing portions of their designs to explore the true area-delay costs. This is entirely reasonable, because so much is different in these designs that leaving them entirely to simulation studies can sweep away subtle details that may have significant impact on the area or clock cycle.

WaveScalar is a recently proposed architecture that is well outside the superscalar mold. Previous work [1] made a "Case for" argument for the architecture that relied upon qualitative comparisons to superscalars and quantitative simulation results. So much is different about WaveScalar compared to superscalars, however – its dataflow instruction set, its memory interface, its distributed ILP substrate – that it really must be shown that the design has reasonable die area requirements and achieves a clock cycle comparable to more conventional processors.

To explore WaveScalar's true area requirements and performance, we built a synthesizable pipelined RTL model of the WaveScalar microarchitecture, called the WaveCache. This model synthesizes with a TSMC 90nm [2] standard cell process. It contains four major components: pipelined processing elements, a pipelined memory interface, a multi-hop network switch and a distributed data cache. These pieces comprise a *cluster*, which is the basic unit of the WaveCache microarchitecture. Clusters are replicated across the silicon die to form the processing chip.

In the process of going from a paper design to a synthesizable RTL model, a large number of design options were explored to meet area, clock cycle, and instructions-per-clock performance targets. This paper will describe the more interesting of these. Where appropriate, we also present results from our cycle-level simulator that illustrate the application performance trade-offs.

By making the proper engineering trade-offs and developing innovations in the RTL implementation, we show that a high

performance WaveCache can be built in current generation 90nm process technology. The processor requires 252$\mathrm{mm}^2$of silicon area. Our tools predict a clock rate of 20.3 FO4 for the execution core and 25 FO4 for the memory interface, leading to a final processor clock of 25 FO4. This clock rate was achieved through aggressively pipelining the microarchitecture originally proposed in [1]. While longer than carefully tuned commercial desktop processors [3], it is faster than other prototypes done in academic settings [4, 5] that used similar tools and design flows.

The design produces performance comparable to superscalars on single-threaded SPEC, 12-54 IPC on multithreaded SPLASH2, and over 100 IPC on dataflow kernels. The same microarchitecture can be scaled down to 18mm^2 for portable systems and up for high-end processors in future process technologies,

In the next section we begin by describing the methodology that is the basis for the microarchitectural designs and evaluation in the rest of this paper. The discussion will elaborate on the tools we use, the salient settings used to produce area data, and how clock rate was estimated. Since executing millions of cycles on an RTL model is prohibitively slow, we also utilize detailed cycle-level simulation to explore instructions-per-cycle performance. This simulation environment is also described.

Following the methodology, section 3 provides a broad overview of the WaveScalar architecture and WaveCache microarchitecture. Sections 4-6 dive deeper, with a detailed presentation of the RTL designs and modeling results for the most significant microarchitectural structures. Section 7 summarizes this work by presenting a couple of additional WaveCache design points and reflecting on our experience with the RTL model. Section 8 presents our conclusions.

2 Methodology

The question of whether to evaluate a proposed architecture by building it has been the subject of much debate [6]. In the past, building a machine to evaluate a new idea (i.e., proof by construction) was popular and many projects undertook this task ([7, 8, 9, 10, 11, 12, 13, 14, 15] are just a few examples from the early 90s). In recent years, however, the costs in both dollars and human effort of fabricating chips in modern process technologies has made building more daunting, and as a result fewer projects [16, 5, 17, 18, 19, 20] have endeavored to build a full implementation. In addition, the availability of cheap, high-performance computers, combined with a good understanding of the basic performance of conventional designs, has made simulation a more viable option.

However, with designs that radically alter basic superscalar structures or start anew, such as WaveScalar, implementing the design answers many critical questions about their viability. Furthermore, the implementation process has the additional benefit of forcing the originally proposed microarchitecure from a high-level design into something that must account for all logic paths and design consequences.

The area and timing (clock cycle time) results we report in this paper are from a tested and synthesized Verilog RTL model of the WaveCache. Our research team is in the process of building a large-scale FPGA-based prototype on which to use this RTL and emulate the full WaveCache system. To gauge application-level performance, we have also constructed a matching cycle-accurate, instruction-level simulator that measures execution time. In the following two subsections we describe the tools used to gather the timing, area, and application performance data described throughout this paper.

2.1 Synthesizable model

Our synthesizable model is written in Verilog. We use the Synopsys DesignCompiler and DesignCompiler Ultra [21] for logical synthesis. The model integrates several Verilog IP models for critical components, such as SRAM cells, arbiters, and functional units.

ASIC design flow: The design rules for manufacturing devices have undergone dramatic changes at and below the 130nm technology node [22]. Issues such as crosstalk, leakage current, and wire delay have required synthesis tool manufacturers to upgrade their infrastructures. The changes have also made it more difficult to draw reliable conclusions from scaling down designs done in larger processes. The data we present in later sections is derived with the design rules and the recommended tool infrastructure of Taiwan Semiconductor Manufacturing Company's TSMC Reference Flow 4.0 [23], which is tuned for 130nm and smaller designs. By using these up-to-date specifications, we ensure, as best as possible, that our results scale to future technology nodes.

As noted by TSMC, designs at and below 130nm are extremely sensitive to placement and routing. Therefore, TSMC recommends against using the delay numbers that are produced after logical synthesis. Instead, they suggest feeding the generated netlist into Cadence Encounter for floorplanning and placement, and then use Cadence NanoRoute for routing [24]. We followed these suggestions in our work. After routing and RC extraction, we recorded the timing and area values. When necessary, the design was fed back into DesignCompiler along with the updated timing information, to recompile the design. The area values presented here include the overhead from incomplete core utilization.

Standard cell libraries: Our design uses the standard cell libraries from the TSMC 90nm process [2]. The 90nm process is the most current process available, and hence represents the best target for extracting meaningful synthesis data. The cell libraries contain all of the logical components necessary for synthesis in both low-power and high-performance configurations. For this study we exclusively use the high-performance cells for all parts of the design, although portions of the design that are not timing critical should later be reimplemented with the low-power cells to reduce power consumption.

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 the other, smaller memory structures. The characteristics (size, delay, etc) of the memory compiler have been explored elsewhere [25].

Timing data: Architects prefer to evaluate clock cycle time in a process-independent metric, fan-out-of-four (FO4). The benefit of using this metric is that the cycle time in FO4 does not change (much) as the process changes. Thus a more direct comparison of designs can be performed.

Synthesis tools, however, report delay in absolute terms (nanoseconds). To report timing data in FO4, we followed the common academic practice [16] suggested in [26] of synthesizing a ring oscillator to measure FO1 and then multiplying this delay by 3. We built an oscillator using the same design flow and standard cells as used in the rest of our design, and measured an FO1 of 16.73ps, which results in an FO4 of 50.2ps. All timing data presented in this paper is reported in FO4 based upon this measurement.

2.2 Cycle-level functional simulation

In conjunction with our Verilog RTL model, we built a corresponding cycle-accurate, instruction-level simulator. Our simulator models each major subsystem of the WaveCache (execution, memory, and network) and is used to explore many aspects in more detail. It also serves to answer basic questions, such as sizing of microarchitecture features and performance impact of contention effects, that arise from the actual design. To drive the simulations, we executed a suite of applications, described below. These applications were compiled with the DEC Alpha CC compiler and then binary translated into WaveCache assembly. These assembly files are compiled with our WaveScalar assembler and these executables are used by our simulator.

Applications: We used three groups of workloads to evaluate the WaveCache, each of which focuses on a different aspect of WaveCache performance. To measure single-threaded performance, we chose a selection of the Spec2000 [27] benchmark suite (*ammp*, *art*, *equake*, *gzip*, *twolf* and *vpr*). To evaluate the effect of multiple threads, we used five of the Splash2 benchmarks: *fft*, *lu-continuous*, *ocean-noncontinuous*, *raytrace*, and *water-spatial*. We chose these subsets of the two suites because they represent a variety of workloads and our binary translator-based tool-chain can handle them. Finally, we use three hand-coded kernels: *mmul* (matrix multiplication), *lcs* (longest common subsequence), and *fir* (a finite input response filter). These kernels exploit novel aspects of the WaveScalar architecture, such as its unordered interface to memory and its ability to spawn extremely fine-grained threads [28]. We run each workload 100 million instructions or until completion.

3 A WaveScalar and WaveCache Overview

This section provides an overview of the WaveScalar architecture and its WaveCache implementation. We confine our discussion of the architecture to those features that provide context for the RTL-level implementation presented later in this paper. A detailed description of the instruction set architecture is presented in [1].

3.1 The WaveScalar Architecture

WaveScalar is a tagged-token dataflow architecture. Like all dataflow architectures (e.g. [29, 30, 31, 32, 33, 34, 12]), 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. Instructions execute after all input operand values have arrived, according to a principle known as the *dataflow firing rule* [29, 30].

WaveScalar executes programs written with conventional von Neumann-style memory semantics (i.e., those composed in languages such as C/C++), correctly ordering memory operations with an architectural technique called *wave-ordered memory* [1] (Section 6 reviews wave-ordered memory in detail). WaveScalar supports traditional dataflow memory semantics (I-structures [35] and M-structures [36]) by a lightweight locking primitive described in [28].

Microarchitecture: From the programmer's perspective, every static instruction in a program binary has a dedicated processing element (PE). Clearly, building so many PEs is impractical and wasteful, so, in practice, we dynamically bind multiple instructions to a fixed number of PEs, and swap them in and out on demand. We say that the PEs *cache* the working set of the application; hence, the microarchitecture that executes WaveScalar binaries is called a *WaveCache*. Figure 1 illustrates how a WaveScalar program can be mapped into a WaveCache. The conflicting goals of the instruction mapping algorithm (which maps dynamically as the program executes) are to place dependent instructions near each other to minimize producer-consumer latency, and to spread independent instructions out in order to utilize resources and exploit parallelism.

```
int *V;
int a, b;
int c, d, r;

r = a*c + b*d;
V[a] = 2*r + d << 2;
```

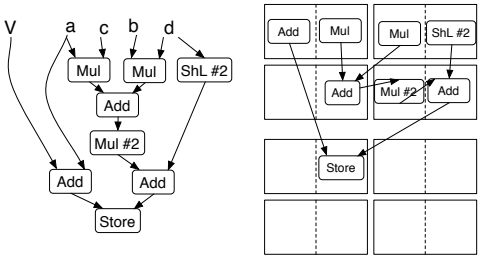

Figure 1: **Three views of code in WaveScalar:** At left is the C code for a simple computation. The WaveScalar dataflow graph is shown at center, and the same graph is mapped onto 2 8-PE domains in the WaveCache substrate at right.

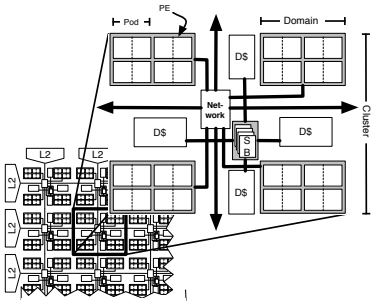

Figure 2: **The WaveCache and cluster:** The hierarchical organization of the WaveCache microarchitecture.

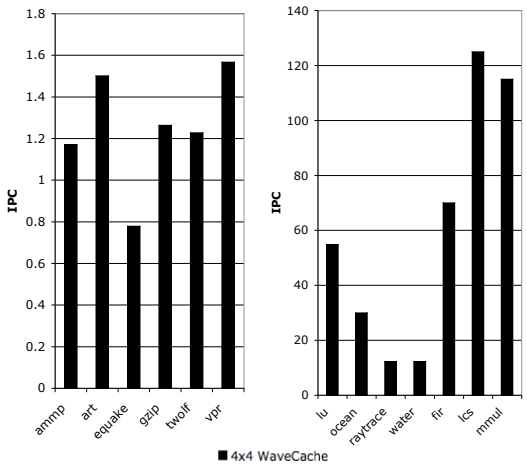

Figure 3: **The baseline performance of a 4x4 WaveCache:** Note the difference in scale for the single-threaded Spec2000 applications and multi-threaded Slash2 and fine-grain applications.

Each PE contains a functional unit, specialized memories to hold operands, and logic to control instruction execution and communication. It also contains buffering and storage for several different static instructions. The PE has a five-stage pipeline, with bypass networks allowing back-to-back execution of dependent instructions at the same PE. Two aspects of the design warrant special notice. First, it avoids a large centralized, associative tag matching store, found on some previous dataflow machines [32]. Second, although PEs dynamically schedule execution, the scheduling hardware is dramatically simpler than a conventional dynamically scheduled processor. Section 4 describes the PE design in detail.

To reduce communication costs within the grid, PEs are organized hierarchically, as depicted in Figure 2. PEs are first coupled into *pods* within a pod PEs snoop each others result networks and share scheduling information. These 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, albeit with a possibly high cache miss rate as instructions are swapped in and out of the small design. To build larger and higher performing machines, multiple clusters are connected by an on-chip network, and cache coherence

WaveCache Capacity	32K static instructions (64 per PE)		
PEs per Domain	8 (4 pods)	Domains / Cluster	4
PE Input Queue	16 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 Caches	32KB, 4-way set associative, 128B line, 4 accesses per cycle	L2 Cache	16 MB shared, 128B line, 16-way set associative, 20 cycle access
Main RAM	1000 cycle latency	Network Switch	2-port, bidirectional

Table 1: Microarchitectural parameters of the baseline WaveCache

is maintained by a traditional, directory-based protocol with multiple readers and a single writer. The coherence directory and the L2 cache are distributed around the edge of the grid of clusters.

The baseline design: The RTL-level model we describe in this paper is a 4x4 array of 16 clusters, each containing a total of 16 pods (32 PEs), arranged 4 per domain. In our 90nm process, each cluster occupies $16\mathrm{mm}^2$, yielding a $263\mathrm{mm}^2$ WaveCache.

In the next three sections, we describe the RTL model of a WaveCache processor composed of 16 clusters in a 4x4 grid. During the design of this model, we chose among many design options, based on the effect they had on delay, area, and application performance. To put the performance trade-offs we discuss in perspective, Figure 3 shows the performance of our baseline design on the workloads we described in Section 2. Table 1 summarizes the baseline configuration. We discuss performance more thoroughly in Section 7.

4 Processing Elements

The WaveCache contains the same overall structures as all computing devices, namely execution, interconnect, and memory resources. We are going to present its microarchitecture using this organization, to give the reader a context in which to view each type of resource. This section focuses on the execution resources; the next section describes how these execution resources communicate with each other; and section 6 describes how they interface with memory.

The execution resources of the WaveCache are comprised of hundreds of pipelined processing elements (PEs). We will present the microarchitecture of the PEs by first describing their function and a broad overview of their pipeline stages. We will then present an example which traces the execution of a short sequence of instructions through this pipeline. Following this example, we describe each pipeline stage in detail.

4.1 A PE's Function

At a high level the structure of a PE pipeline resembles a conventional five-stage, dynamically scheduled execution pipeline. The biggest difference between the two is that the PE's execution is entirely data-driven. Instead of executing instructions provided by a program counter, as you find on von Neumann machines, values arrive at a PE destined for a particular instruction. These values trigger execution – the essence of dataflow execution. A pre-decoded instruction is fetched from a local instruction store in the PE and, when all instruction inputs are available, the instruction executes and sends its result to trigger the execution of other instructions.

The five pipeline stages of a PE are:

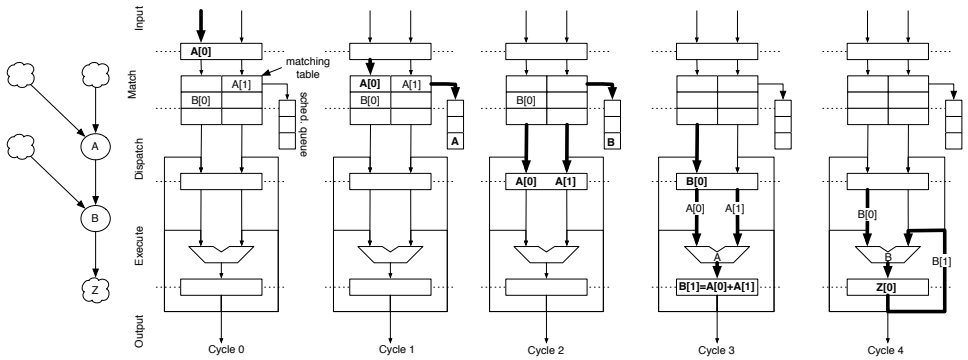

Figure 4: **The flow of operands through the PE pipeline and forwarding networks:** The figure is described in detail in the text.

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

2. MATCH: Operands enter the operand *matching table*. The matching table determines which instructions are now 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 destination of the dispatched instruction is local, this stage speculatively issues the consumer instruction to the scheduling queue.

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

5. OUTPUT: Instruction outputs are sent via the output bus to their consumer instructions, either at this PE or a remote PE.

The pipeline design includes bypass paths that allow operands to move directly from the beginning of EXECUTE to the beginning of EXECUTE. This bypass network, combined with hardware scheduling, enables back-to-back execution of dependent instructions.

Figure 4 illustrates how instructions from a simple dataflow graph flow through the pipeline and how their execution affects the matching table and scheduling queue. It also illustrates how the bypass network allows two instructions A and B to execute on consecutive cycles. In this sequence, 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: Operands $A[0]$ 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 into 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 scheduling 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.

Cycle 5 (not shown): OUTPUT will send B's output to Z.

This example serves to illustrate the basic mechanics of PE operation. We will now describe each stage in detail, as well as the design trade-offs involved in each.

4.2 INPUT

Each cycle, INPUT monitors the incoming operand busses. In our RTL model there are 10 busses: 1 is the PE's output bus, 7 originate from other PEs in the same domain, 1 is the network bus and 1 is the memory interface. INPUT will accept inputs from up to four of these busses each cycle. If more than four arrive during one cycle, an arbiter selects among them; rejected inputs are retransmitted by their senders. Four inputs is a reasonable balance between performance and design complexity/area. Due to the banked nature of the matching table (see below), reducing the number of inputs to three had no practical area-delay benefit. Two inputs, however, reduced application performance by 5% on average, but by 15-17% for some applications (*ammp* and *fir*). Doubling the number of inputs to eight increased performance by less than 1% on average.

As previously mentioned in Section 3, WaveScalar is a tagged token dataflow machine. This means all data values carry a tag that differentiates dynamic instances of the same value. Tags in WaveScalar are comprised of two fields: a THREAD-ID and a WAVE-NUMBER. Since each PE can hold multiple static instructions, messages on the busses also carry a destination instruction number. [1] INPUT computes a simple XOR hash of the THREAD-ID, WAVE-NUMBER, and destination instruction number for each operand, which is used to index the matching table. INPUT then places the (up to four) operands it has selected, along with their hashes, into its pipeline register for MATCH to process in the next clock cycle.

Neglecting domain wiring overhead, which will be accounted for later in this section, INPUT's actual logic consumes 8.3% ($0.03\mathrm{mm}^2$) of the PE's area. It achieves a clock rate of 13.7 FO4 in isolation, which is significantly shorter than the other stages. However, the rest of the clock period is taken up by delay in the intra-domain interconnect.

4.3 MATCH

The next two pipeline stages comprise the operand tag matching and instruction dispatch logic. Implementing these operations cost-effectively is essential to an efficient dataflow design and has historically been an impediment to more effective dataflow execution [32]. The key challenge in designing the WaveCache matching table is emulating a potentially infinite table with a much smaller physical structure. This problem arises because WaveScalar is a dynamic dataflow architecture, and places no limit on the number of dynamic instances of a static instruction with unconsumed inputs.

To address this challenge, the matching table is a specialized cache for a larger in-memory matching table, a common dataflow technique [32, 31]. MATCH writes operands into the matching table, and DISPATCH reads them out. The table is separated into three columns, one for each potential instruction input. Associated with the matching table is a *tracker board*, which holds the operand tag, consumer instruction number, *presence bits* which denote which operands have arrived, and a *pin bit* which indicates which instructions have all of their operands and are ready to execute.

When new operands arrive from INPUT, the PE attempts to store each of them in the matching table, using the hash as the

[1] In addition, messages contain a destination operand number, used to determine which input queue to place the value in.

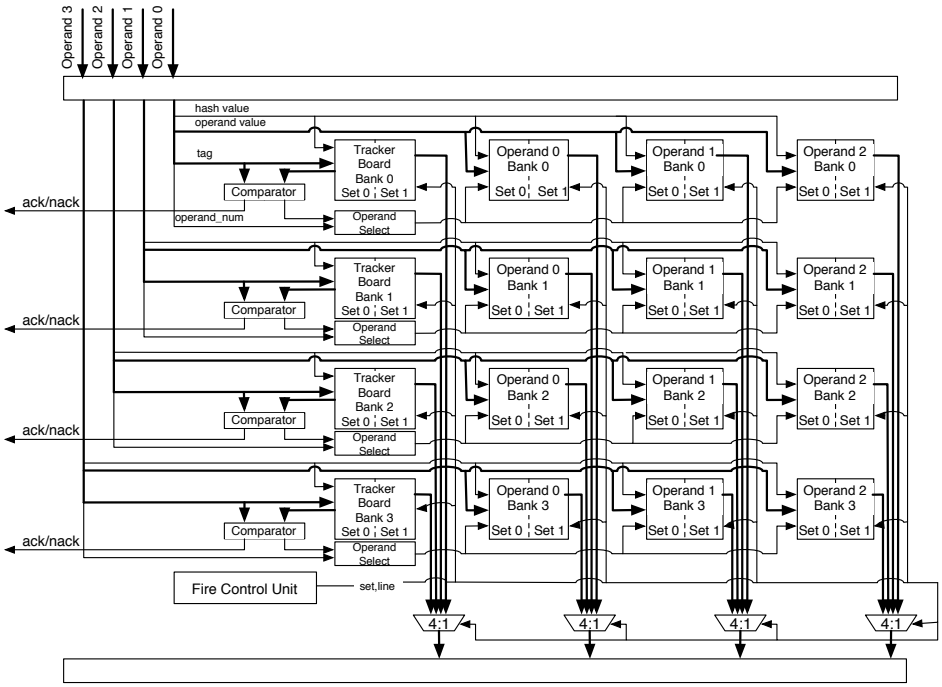

Figure 5: **Matching table detail:** The matching table uses a large number of small, single-ported SRAMS to allows up to four operands to be written each cycle. The tracker board detects when instructions are ready to fire.

index. For each operand, there are four possibilities: (1) An operand with the same tag has already arrived, so there is a space waiting for it in the matching table. (2) No other operands with the same tag have arrived, and the line is unoccupied. In this case, MATCH allocates the line to the new operand and updates the tracker board. (3) The line is occupied by the operands for another instruction. In this case, the PE rejects the message, and waits for the sender to retry. After several retries, operands resident in the matching table are evicted to memory, and the newly empty line is allocated to the new operand. (4) The line is occupied by the operands for another instruction which is pinned to the matching table. This occurs when the instruction is ready to, but has not yet executed. As in case (3), the message is rejected and will be resent. After four retries the new operand is written to memory. Scenarios (3) and (4) are *matching tables misses*.

In parallel with updating the matching table, MATCH checks the presence bits to see if any of the operands that arrived in INPUT were the last ones needed to allow an instruction to execute. If this is the case, MATCH pins the corresponding line in place and adds its matching table index and tag to the scheduling queue (described in the next section).

While the average occupancy of the matching table is low, we found it critical to have at least 16 entries to handle bursty behavior. Reducing the number of entries to 8 dropped performance on average by 23%. Doubling to 32 added almost no gain to applications written in C, but increased performance on fine-grained dataflow kernels by 36%. Because this configuration consumes substantially more area (see below) and provides limited improvement on C-based applications, we choose 16 entries for our RTL implementation. This application-class variation, however, suggests that designers will want to tune this parameter, depending on the target market.

Since the matching table is a cache, we can apply traditional caching optimizations, such as associativity and banking, to reduce area requirements, miss rate, and miss penalty. The basic design is 2-way set associative, and each way is banked by 4,

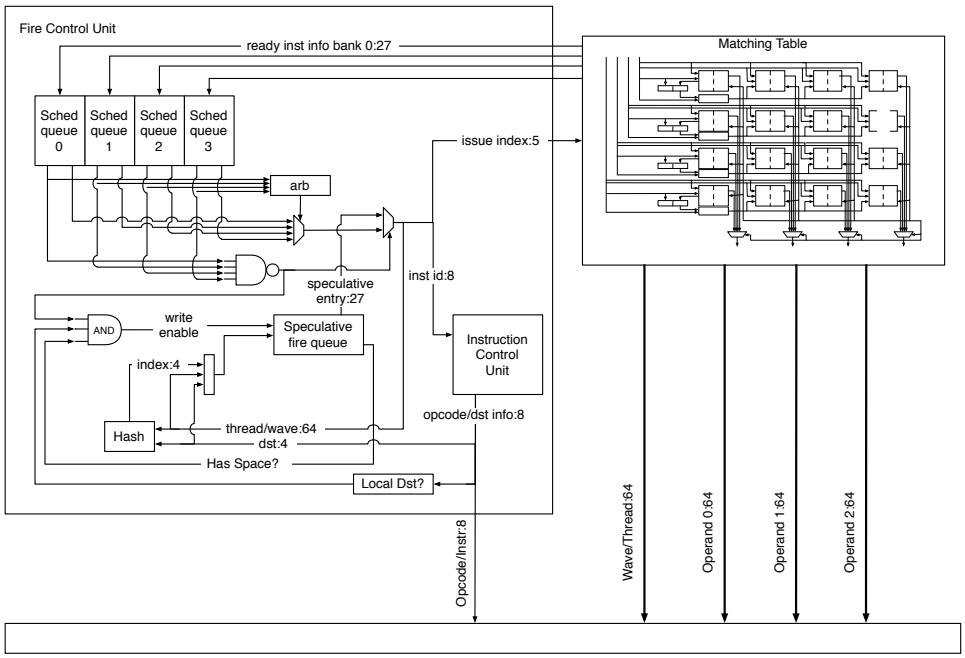

Figure 6: **The dispatch stage:** DISPATCH schedules execution on the ALU and is in charge of allowing dependent instruction to execute on consecutive cycles.

to increase read/write port availability. Given the three operand queues and tracker board, this means the entire design requires 32 small SRAMs (4 tables, 2 sets/table, 4 banks/set, as shown in Figure 5, bottom), each of which contains 2 matching table entries. SRAMs for the first two operands are 64 bits wide. The third operand is used only for single-bit operands (control bits), so its SRAMS are 1 bit. Tracker board SRAMs are 76 bits.

The figure shows the data paths between the SRAMs in detail. Operand messages from INPUT arrive at the top. Data values flow down to the operand arrays, while tag information travels to the tracker board. A comparator determines whether a line has previously been allocated to an operand; the hash value, operand select, and tracker board pick the line, bank and "way" where the operands should reside. Bank conflicts in the matching table are handled by rejecting the input.

RTL synthesis of MATCH shows that, in isolation, MATCH hardware consumes $0.1\mathrm{mm}^2$, 29.8% of total PE area, and achieves a clock cycle of 20.3 FO4. Doubling the input queue size gives a near linear increase in area ($0.17\mathrm{mm}^2$ or 39% of the PE) – a 20% increase in overall PE size, and 5% increase in delay. MATCH and DISPATCH are the longest stages in the PE, so increases in queue size should be considered with care.

4.4 DISPATCH

The DISPATCH (Figure 6) stage and the fire control unit (FCU) are in charge of scheduling instructions for execution. In the simplest dispatching case, the FCU removes the first entry from the scheduling queue, reads the corresponding line from the matching table, and passes the operands to EXECUTE for execution. This behavior is sufficient for correct execution, but does not allow dependent instructions to execute on consecutive clock cycles.

To achieve back-to-back execution of dependent instructions, we provide bypassing paths that send results from the end of EXECUTE directly back to the beginning of EXECUTE. In addition, the FCU can speculatively issue a consumer of the result, readying it to use the newly produced result on the next cycle. In particular, when the FCU selects an entry from the

scheduling queue, it accesses the instruction store to determine which, if any, of the instruction's consumers reside at the same PE. If there is a local consumer, the FCU computes the index of its line in the matching table and inserts it into a special scheduling queue, called the *speculative fire queue*.

Placing a consumer instruction in the speculative fire queue is speculative because the FCU cannot tell whether the producer's result will allow it to fire (i.e., whether the instruction's other operands already reside in the matching table). In the example in Figure 4, although the FCU knows that A will produce operand $B[1]$, it does not know if B's second input, $B[0]$, is present in the matching table. Operand availability is resolved in EXECUTE, where the speculative instruction's tag from the matching table (the unknown operand, sent to EXECUTE when the consumer is dispatched) is compared to the tag of the producer's result (the known operand, just computed). If they match, and if the presence bits match the required operand signature bits, the consumer instruction executes successfully and the matching table entry is cleared. If not, then the result is squashed, and the matching cache entry is left unchanged.

DISPATCH gives higher priority to keeping a PE busy than dispatching dependent instructions back-to-back. Therefore, it will usually choose for execution nonspeculative instructions over speculative. In particular, if there are enough nonspeculative instructions in the scheduling queue to allow a producer's result to flow from OUTPUT back to MATCH (where it will be placed in the matching table and the match table logic will determine whether the consumer should fire), DISPATCH will choose the nonspeculative instructions. Otherwise, it will gamble that all the consumer's operands have arrived and dispatch it.

The scheduling queue size is 16 entries, chosen to be equivalent to the matching table, thus simplifying the design. The speculatively scheduled queue slot is maintained in a separate register.

The final piece of the FCU is the Instruction Control Unit (ICU), which contains the PE's decoded static instructions: their opcodes, the consumers of their results, and immediate values. The ICU in the RTL design holds 64 decoded static instructions, each 59 bits. Decreasing the number of instructions to 32 impacts performance by 23% on average; doubling it to 128 increases performance by only 3% but also increases ICU area by 120% and PE area by 55% and cycle time by 4%. Nevertheless, our results indicate that designers of small WaveCache's (one or a small number of clusters) should choose the larger design.

DISPATCH shares a large portion of its logic with MATCH. The separate hardware is comprised of the ICU, the scheduling queue, and the control logic. These added components require $0.17\mathrm{mm}^2$(49% of the PE area), nearly all of which is in the ICU. DISPATCH has the same delay as MATCH(20.3 FO4).

4.5 EXECUTE

EXECUTE (Figure 7) handles three different execution scenarios: (1) The usual case is that all operands are available and the output queue can accept a result. The instruction is executed, the result written to the output queue, and the line in the matching table is unpinned and invalidated. (2) A speculative instruction, some of whose inputs are missing, was dispatched. In this case the result is squashed, and the matching table line is unpinned but *not* invalidated. (3) No space exists in the output queue. In this case, EXECUTE stalls until space is available.

In addition to a conventional functional unit, EXECUTE contains a tag-manipulation unit that implements WaveScalar's tag manipulation instructions [28] and logic for handling its data steering instructions. PEs are non-uniform. In our current

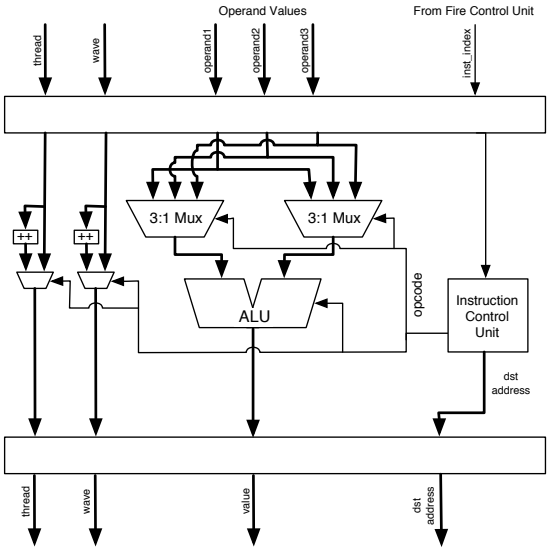

Figure 7: **The execution stage:** The PE's execute stage contains a single general purpose ALU that accepts three input operands and implements the WaveScalar instruction set.

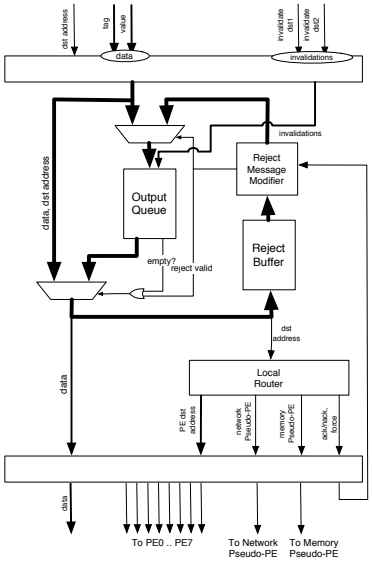

Figure 8: **The output interface:** OUTPUT distributes the PE's outputs to their consumers. Outputs reside in the output queue until they are sent. In addition, rejected operands await retransmission over the intra-domain interconnect (Section 5) in the reject buffer.

design 6 PEs compute integer instructions only. These require 0.02mm^2 (5.7% of the PE). Two PEs per domain contain a floating point unit in addition to the integer core. These FPU-enabled PEs require an additional 0.12mm.

4.6 OUTPUT

OUTPUT sends a result from the ALU to the consumer instructions that require it. Figure 8 shows its design. OUTPUT contains a 4-entry output queue that is connected directly to the PE's output buffer. Values can enter the output queue either from EXECUTE or from the reject buffer (explained below). If the output queue is empty, incoming values go directly to the output buffer. The precise size of the output queue has little effect on performance – four entries is sufficient. The reason it tends not to influence performance is that result values normally flow uninterrupted to their destination PEs.

The output buffer broadcasts the value on the PE's broadcast bus. In the common case, the consumer PE within that domain accepts the value immediately. It is possible, however, that the consumer cannot handle the value that cycle and will reject it. ACK/NACK signals require four cycles for the round trip. Rather than have the data value occupy the output register for that period, the PE assumes it will be accepted, moving it into the 4-entry *reject buffer*, and inserts a new value into the output buffer on the next cycle. If an operand ends up being rejected, it is fed back into the output queue to be sent again to the destinations that rejected it. If all the receivers accept the message, the reject buffer discards the value. When rejected messages are going from the reject buffer to the output queue, any message from the execution unit bypasses the output queue to avoid queueing two messages on the same cycle. We will describe this process in more detail in Section 5.2.

Each instruction has its consumer instruction locations stored in the instruction cache. The destinations can either be to memory, or to up to two other PEs. Each destination has a valid bit that is cleared whenever the destination PE accepts the message. This can happen either through the standard output network, or when PEs in the same pod successfully execute a

speculatively scheduled instruction. The output queue stops sending the message when all destination bits are clear.

Since there is no determined length of time that an entry can sit in the matching cache, there must be a mechanism for preventing messages from cycling through the reject buffer enough times to affect the sender's performance. To handle this, the sender keeps a 2-bit counter of the number of times that the message has been rejected. When this counter reaches its maximum value, the sender requests that the receiver forcefully accept the message. When the receiver gets a forced accept request, it rejects the message, but places the entry that is blocking the message into the scheduling queue to fire. Instead of firing normally, the entry is sent through the pipeline without modifications to its tag or data. This entry will then travel through the pipeline in the standard manner, but instead of going to its destination, it goes to the memory pseudo-PE with a special flag to indicate that the message should be sent back later. The memory pseudo-PE holds a table of entries that have been sent to the L1 cache that need to be resent to the domain, and retrieves those entries later. In the special case that two operands are stalled, then the fire control unit will send each operand in a separate message. This mechanism requires very little extra logic to implement, and guarantees that each message will eventually make it to the receiver.

The output stage consumes 9% of the PE's area. It achieves a clock rate of 17 FO4 in isolation, and the remainder its clock cycle is devoted to wire delay in the intra-domain interconnect.

4.7 PE area and timing

In total, each PE consumes $0.36\mathrm{mm}^2$, and all together comprise 87% of total chip area. The matching table stage in the PE is the critical path (20.3 FO4) for both the PE and the domain. Within MATCH, the longest path is the one that updates the scheduling queue. This path depends on a read/compare of the matching table.

In addition to the 8 PEs, each domain contains two psuedo-PEs (called MEM and NET) that serve as portals to the memory system and PEs in other domains and other clusters. Each pseudo-PE contains buffering for 64 messages. The NET and MEM pseudo-PEs are $0.08\mathrm{mm}^2$and $0.06\mathrm{mm}^2$, respectively.

An entire domain occupies $3.6\mathrm{mm}^2$. In order to estimate the area of the domain exclusive of its intra-domain interconnect (described in the next section), we synthesized the PEs in isolation and compared this to the total domain area after processing with Cadence Encounter. Using this estimate we found the domain interconnect to be 8.6% of the total domain size.

5 The Network

The previous section described the execution resource of the WaveCache, the PE. This section will detail how PEs on the same chip communicate. PEs send and receive data using a hierarchical on-chip interconnect. There are four levels in this hierarchy: intra-pod, intra-domain, intra-cluster and inter-cluster. The first three of these networks are illustrated in Figure 9, which depicts a single cluster. The fourth network, the inter-cluster network, is a dynamically routed packet network that connects the clusters. While the purpose of each network is the same – transmission of instruction operands and memory values – the design varies significantly across them. We will describe salient features of these networks in the next four subsections.

5.1 PEs in a Pod

The first level of interconnect, the intra-pod interconnect, allows two PEs to share their bypass networks and scheduling information. Merging a pair of PEs into a pod provides lower latency communication between them than using the intra-

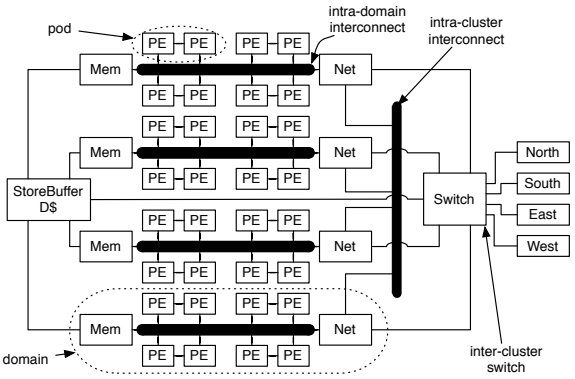

Figure 9: **The cluster interconnects:** A high-level picture of the interconnects within a cluster.

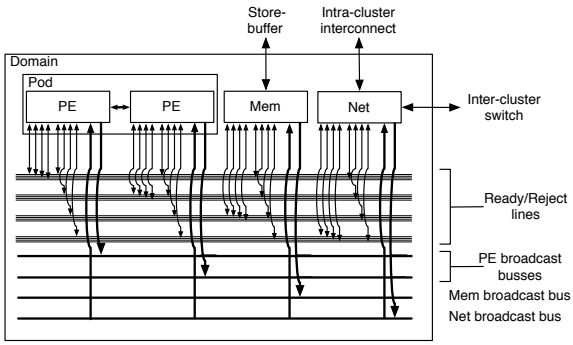

Figure 10: **The intra-domain interconnect:** For space reasons, the interconnect for a 2-PE (1 pod) domain is shown. (Our design includes 8-PEs.) The thick horizontal lines are the broadcast busses for each PE and the network and memory interfaces. The fine lines are the ACK and NAK signals. Communication paths through the interconnect move down to a broadcast bus, across, and then up to their destination.

domain interconnect (see below).

While PEs in a pod snoop each others bypass networks, all other aspects of a PE remain partitioned – separate matching tables, scheduling and output queues, etc. The intra-pod network transmits data from the execution units, and transmits instruction scheduling information from the Fire Control Units.

Currently, our RTL model is implemented with two PEs per pod. Our simulations show that this design is 5% faster on average than PEs in isolation and up to 15% faster for *vpr* and *ammp*. Increasing the number of PEs in each pod would further increase IPC, but since DISPATCH is already the longest stage in the PE, it would have a detrimental effect on cycle time.

5.2 The intra-domain interconnect

PEs communicate over an intra-domain interconnect, shown in detail in Figure 10. Its interface to both PEs and pseudo-PEs is identical.

The intra-domain interconnect is broadcast-based. Each of the eight PEs has a dedicated 164-bit result bus that carries a single data result to the other PEs in its domain. Each pseudo-PE also has a dedicated 164-bit output bus. PEs and pseudo-PEs communicate over the intra-domain network using a garden variety ACK/NACK network. We illustrate the timing of this network in our design with an example (Figure 11). In this example PE0 is trying to send $D0$ to PE1 and PE2 and $D1$ to PE1.

Cycle 0: PE0 sends $D0$ to PE1 and PE2. The OUTPUT stage at PE0 prepares the message and broadcasts it, asserting the PE1 and PE2 receive lines.

Cycle 1: PE0 sends $D1$ to PE1, which reasserts its receive line. At PE1 and PE2, INPUT processes $D0$ and sends it to MATCH.

Cycle 2: PE0 goes idle. INPUT at PE1 receives $D1$ and sends it to MATCH. MATCH of PE2 detects a matching table conflict for $D0$ and asserts the NACK signal. PE1 does not have a conflict and, by not asserting NACK, accepts the message.

Cycle 3: The interconnect delay.

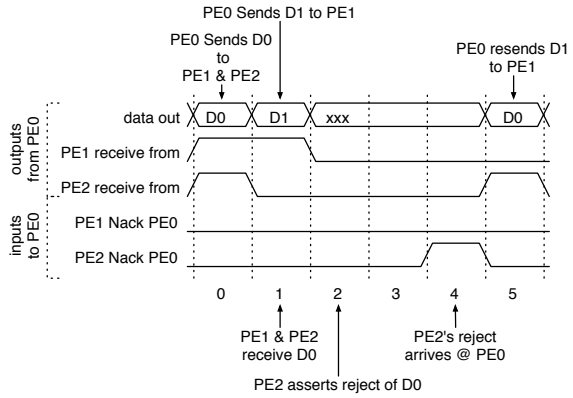

Figure 11: **ACK/NACK timing:** The timing of a simple transaction between PE0 and PE1

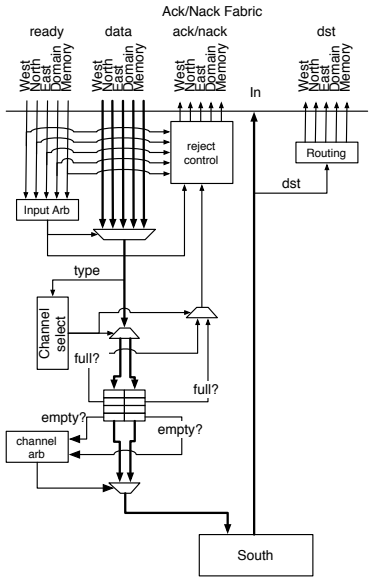

Figure 12: **The southern port of the inter-cluster switch:** Incoming messages are routed toward their destination (West, North, East, PEs, or Store buffer/L1 Cache). Depending on the type of an outgoing message the switch uses one of two virtual channels to route the message to the south.

Cycle 4: PE0 receives the NACK signal from PE2 for $D0$.

Cycle 5: PE0 notes that PE1 accepted $D1$ and attempts to retry sending $D0$ to PE1.

There are two advantages of using ACK/NACK flow control for this network. The first is a large reduction in area. There are 10 inputs to each PE, and adding a two-entry buffer to each input would require 2868 bits of buffering at each receiver. Instead, we only use 169 bits of buffering at the sender. Second, ACK/NACK flow control allows messages to bypass the rejected messages. The consequence of these advantages are a lower clock rate and sustained bandwidth.

The downside, however, is that rejected messages take far longer to process. In our experiments, we found that on average fewer than 1% of messages were rejected. As there is only one ALU per PE, provisioning the network to send more than one result per cycle is useful only for processing these relatively few rejected messages. Widening the PE broadcast busses to transmit two results increased performance negligibly and significantly increased the complexity of the PEs' input and output interfaces.

5.3 The intra-cluster interconnect

The intra-cluster interconnect provides communication between the four domains' NET pseudo-PEs. It also uses a ACK/NACK network similar to that of the intra-domain interconnect, with some additional buffering. An extra pipeline stage is added to the network to account for wire delay. The pseudo-PEs occupy only 8% of the domain area. Synthesized in isolation, they pass timing at 20 FO4 with considerable slack to spare (i.e., they can be clocked faster).

5.4 The inter-cluster interconnect

The inter-cluster interconnect is responsible for all long-distance communication in the WaveCache. This includes operands traveling between PEs in distant clusters and coherence traffic for the L1 caches.

Each cluster contains an inter-cluster network switch, each of which routes messages between 6 input/output ports. Four of the ports lead to the network switches in the four cardinal directions, one is shared among the four domains' NET pseudo-PEs, and one is dedicated to the store buffer and L1 data cache.

The inter-cluster interconnect uses a simple dynamic routing switch. Each switch has 6 input/output ports, each of which supports the transmission of up to 2 operands. Its routing follows a simple protocol: the current buffer storage state at each switch is sent to the adjacent switches, which receive this information a clock cycle later. Adjacent switches only send information if the receiver is guaranteed to have space.

The inter-cluster switch provides two virtual channels that the interconnect uses to prevent deadlock [37]. Figure 12 shows the details of one input/output port of the inter-cluster switch. Each output port contains two 8-entry output queues (one for each virtual network). In some cases, a message may have two possible destinations (i.e., North and West if its ultimate destination is to the northwest). In these cases the router randomly selects which way to route the message.

The network carries messages that are 164 bits wide and include a destination location in the grid. 64 bits are used for data, and 64 bits for tag; the additional bits are for routing. The destination routing includes the following elements: destination cluster x and y (4 bits each), destination domain (2 bits), destination PE (3 bits), destination virtual slot number (6 bits), and destination operand number (2 bits). Memory messages are also routed over this network, and share routing bits with those used for sending operands. Memory messages are routed with the cluster position, sequence tag information [1] (15 bits) and store buffer number (2 bits).

In the TSMC process there are 9 metal layers available. This means the long distance inter-cluster wires sit above the main cluster logic, minimizing the area impact of the switches. Each cluster switch requires $0.34\mathrm{mm}^2$ and achieves a clock cycle of 19.9 FO4. In aggregate, the network switches account for 2% of the entire die.

5.5 Network traffic

One goal of the WaveCache interconnect is to isolate as much traffic as possible in the lower layers of the hierarchy (e.g., within a PE, a pod, or a domain), and rely on the upper levels only when absolutely necessary. Figure 13 shows the division of traffic among different layers of the hierarchy. On average 28% of network traffic travels from a PE to itself or the other PE in its pod, 48% of traffic remains within a domain and only 2.2% needs to traverse the inter-cluster interconnect. Fine-grain applications require more inter-cluster traffic (33% of operands). This reflects the increased synchronization overhead required for fine-grain threading.

The graph also shows the division between operand data and memory/coherence traffic. Memory traffic accounts for 12% of messages on average. For the Spec2000 applications less than 1% of those messages leave the cluster, because the instruction working set for each of these applications fits within a single cluster. Data sharing in the Splash2 benchmarks increases inter-cluster memory traffic to 17% of memory traffic, but still only 0.4% of total network traffic – everything else is local.

These results demonstrate the scalability of communication performance on the WaveCache. Applications that require

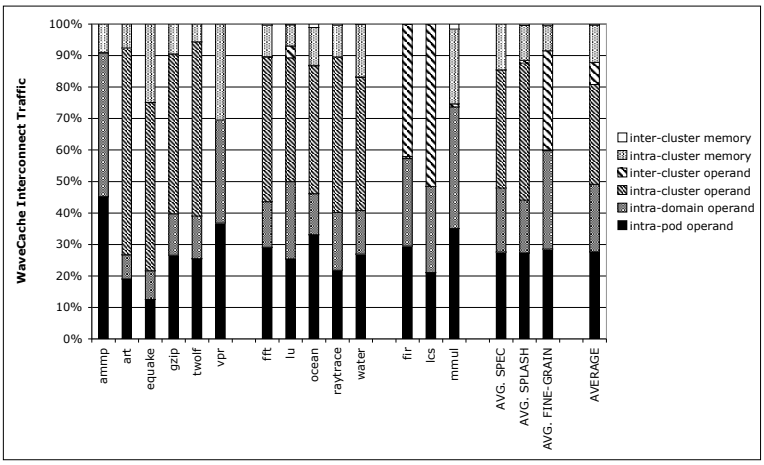

Figure 13: **The distribution of traffic in the WaveCache.:** The vast majority of traffic in the WaveCache is confined within a single cluster and, for many applications, over half travels only over the intra-domain interconnect.

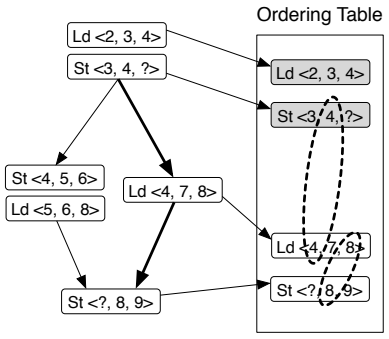

Figure 14: **Annotating memory operations.:** A simple wave's control flow graph (left), showing the memory operations in each basic block, their ordering annotations, and how the annotations allow the store buffer to reconstruct the correct order (right). The darker arrows show the executed path.

only a small patch of the WaveCache, such as Spec, can execute without ever paying the price for long distance communication.

6 The Memory Subsystem

Waves and wave-ordered memory enable WaveScalar to execute programs written in imperative languages, such as C, C++, or Java [38, 39], by providing the well-ordered memory semantics these languages require. The hardware support for wave-ordered memory lies in the WaveCache's store buffers. To motivate the purpose and design choices of the store buffer implementation, we briefly review the concept of wave-ordered memory, first introduced in [1]. After examining the store buffer design, we briefly describe the L1 data cache, which is a conventional lockup-free cache [40] that uses a standard directory-based cache coherence protocol [41].

6.1 Waves and Wave-ordered Memory

WaveScalar is a tagged token dataflow machine. It supports execution of applications written in mainstream imperative languages through the use of a special memory interface, *wave-ordered memory*. The key difference in implementing the hardware for this interface, as compared to a conventional store buffer or load/store queue, is the order in which memory operations fire. Instead of being sequenced by an instruction fetch mechanism, it is under direct program control. A detailed description of this interface is described in [1]; here we give a brief review, in order to provide context for the microarchitectural design.

When compiling a WaveScalar program, a compiler breaks its control flow graph into pieces called *waves*. The key properties of a wave are: (1) its instructions are partially ordered (i.e., it contains no back-edges), and (2) that control enters at a single point. The compiler uses the control flow graph and the instruction order within basic blocks to annotate each memory operation with (1) its position in its wave, called a *sequence number*, and (2) its execution order (predecessor and

successor) relative to other memory operations in the same wave, if they are known. Otherwise, they are labeled with '?'.

During program execution, the memory system (in our implementation, a store buffer) uses these annotations to assemble a wave's loads and stores in the correct order. Figure 14 shows how the wave-ordering annotations allow the store buffer to order memory operations and detect those that are missing. Assume the load with sequence number 7 is the last instruction to arrive at the store buffer. Before its arrival, the store buffer knows that at least one memory operation between operations 4 and 8 is missing, because 4's successor and 8's predecessor are both '?'. As a result, it cannot execute operation 8. Operation 7's arrival provides the link between operations 4 and 8, allowing the store buffer to execute both 7 and 8.

6.2 Store Buffers

The store buffers, one per cluster, are responsible for implementing the wave-ordered memory interface that guarantees correct memory ordering. To access memory, processing elements send requests to their local store buffer via the MEM pseudo-PE in their domain. The store buffer will either process the request or direct it to another buffer via the inter-cluster interconnect. All memory requests for a single *dynamic* instance of a wave (for example, an iteration of an inner loop), including requests from both local and remote processing elements, are managed by the same store buffer.

To simplify the description of the store buffer's operation, we denote $R.pred$, $R.seq$, and $R.succ$ as the wave-ordering annotations for a request R. We also define $next(R)$ to be the sequence number of the operation that actually follows R in the current instance of the wave. $next(R)$ is determined either directly from $R.succ$ or is calculated by the wave-ordering memory, if $R.succ$ is '?'.

The store buffer contains four major microarchitectural components: an *ordering* table, a *next table*, an *issued* register, and a collection of *partial store queues*. Store buffer requests are processed in three pipeline stages: MEMORY-INPUT writes newly arrived requests into the ordering and next tables. MEMORY-SCHEDULE reads up to four requests from the ordering table and checks to see if they are ready to issue. MEMORY-OUTPUT dispatches memory operations that can fire to the cache or to a partial store queue (described below). We detail each pipeline stage of this memory interface below.

MEMORY-INPUT accepts up to four new memory requests per cycle. It writes the address, operation, and data (if available, for stores) into the ordering table at the index $R.seq$. If $R.succ$ is defined (not '?'), the entry in the next table at location $R.seq$ is updated to $R.succ$. If $R.prev$ is defined, the entry in the next table at location $R.prev$ is set to $R.seq$.

MEMORY-SCHEDULE maintains the issued register, which points to the next memory operation to be dispatched to the data cache. It uses this register to read four entries from the next and ordering tables. If any memory ordering links can be formed i.e., next table entries are not empty, the memory operations are dispatched to MEMORY-OUTPUT and the issued register is advanced. The store buffer supports the decoupling of store-data from store-addresses. This is done with a hardware structure called a *partial store queue*, described below. The salient point for MEMORY-SCHEDULE, however, is that stores are sent to MEMORY-OUTPUT even if their data has not yet arrived.

MEMORY-OUTPUT reads and processes dispatched memory operations. Four situations can occur: (1) The operation is a load or a store with its data and is sent to the data cache. (2) The operation is a load or a store and a partial store queue exists for its address. The memory operation is sent to the partial store queue. (3) The memory operation is a store, its data has not yet arrived, and no partial store queue exists for its address. In this case a free partial store queue is allocated and the store is sent to it. (4) The operation is a load or a store, but no free partial store queue is available or the partial store queue is full.

The operation is discarded and the issued register is rolled back.

Figure 15 illustrates the store buffer logic and structures needed to order a single wave of memory requests. The ordering table has 32 entries; each is composed of four banks that are interleaved to allow four consecutive entries to be read or written each cycle. The ordering table is 130 bits wide, large enough to hold an address and the memory request opcode. The next request table has 32 entries and is 5 bits wide and tracks $next()$ information for the wave.

In our design, each store buffer contains 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. Doubling the number of partial store queues increases performance by only 9% on average, while halving the number reduces it by 5%.

Each store buffer requires $0.6\mathrm{mm}^2$ to implement. Four of these occupy $2.4\mathrm{mm}^2$per cluster. Of this, the partial store queues occupy $0.02\mathrm{mm}^2$. Our design achieves a clock speed of 25 FO4. It is the slowest of any component in our design and sets the clock rate for the device [2].

6.3 Caching and coherence

The rest of the WaveCache's memory hierarchy comprises a 32KB, four-way set associative L1 data cache at each cluster, and a 16MB L2 cache distributed along the edge of the chip (16 banks in a 4x4 WaveCache). A directory-based multiple reader, single writer coherence protocol keeps the L1 caches consistent. All coherence traffic travels over the inter-cluster interconnect.

We have explored larger and smaller size caches. The data is largely commensurate with what one observes in traditional microprocessors; more cache helps performance. The baseline design, with 32KB of cache requires $0.4\mathrm{mm}^2$ per cluster to implement. We expect that hardware designers will choose an appropriate cache size, depending upon their application mix and area constraints, as they do with all processors.

The L1 data cache has a 3-cycle hit delay (2 cycles SRAM access, 1 cycle processing), which can be overlapped with the store buffer processing for loads. The L2's hit delay is 14-30 cycles depending upon address and distance to a requesting cluster. Main memory latency is modeled at 1000 cycles. In our RTL model, the L1 caches occupy 4% of the cluster's area. We assume the L2 is off-chip. Doubling the size of the L1 data caches improves performance by only 3%. Additional cycle delays of larger caches begin to appear only at the 128K (1 additional cycle), and 256K (2 additional cycles) sizes. Shrinking the data cache to 16K has negligible effect on access time.

7 Discussion

7.1 Scaling the WaveCache

The previous three sections described the microarchitectural implementation of the WaveCache. We tuned and built the RTL around the "baseline" WaveCache described in Section 2. That design requires $252\mathrm{mm}^2$in a 90nm process. As with all processors, however, its core memory sizes and bandwidths can be tuned for different market segments. We briefly describe two ends of the design space. The performance and configuration of these alternatives are depicted in Figure 16. For each WaveCache size, we ran the application with the optimal number of threads for that number of clusters.

At the low-end is a $18\mathrm{mm}^2$ single-cluster WaveCache. Our results show that this small WaveCache achieves essentially

[2] Our current effort is geared towards further tuning this component. We expect to hit a faster clock rate target for any final version of this paper.

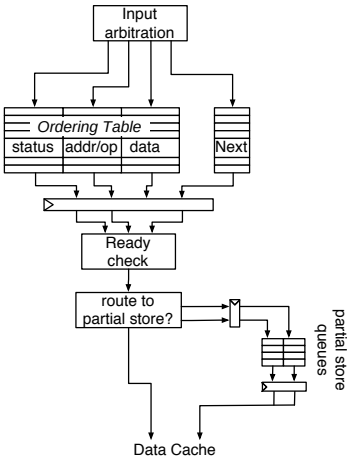

Figure 15: **Store buffer microarchitecture::** The store buffer manages memory requests originating from the execution fabric.

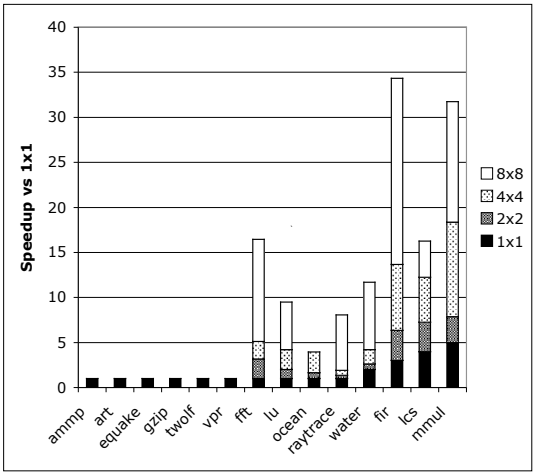

Figure 16: **WaveCache size:** Increased WaveCache size has no effect on Spec2000 applications, because they fit easily in a single cluster. Multithreaded applications, however, benefit substantially from additional clusters.

the same performance on single-threaded C applications as the full-size WaveCache. Obviously, it does not perform as well on parallel applications, such as Splash-2, because there are fewer execution resources available and, therefore, insufficient room to support multiple concurrent threads.

At the high-end, a large 8x8 grid of clusters is possible for future WaveCaches. Such a design does not fit in 90nm technology, but becomes practical at the 45nm technology node, where we estimate that it is roughly [3] $250\mathrm{mm}^2$. This design does little for single-threaded C applications, but increases Splash-2 performance by 2.8 times and fine-grained dataflow kernel performance by a factor of 2 relative to the 4x4 baseline configuration.

7.2 General insights from the RTL model

Few research projects choose to implement and synthesize their proposals. Having gone through this process for WaveScalar, we conclude with a few remarks about our experience that other researchers may find helpful when contemplating designing an RTL implementation.

Designs are limited by tools: While CAD tool vendors, such as Synopsys and Cadence, provide low- to no-cost academic access to their most sophisticated tools, these tools have their limits. One example where the WaveScalar design changed significantly on the microarchitectural level due to the limitations of tools is in the various memory structures. Architecturally, it is simple to imagine (and simulate) memory structures with multiple write ports or content-addressability. However, without doing partially custom designs or utilizing additional, usually unavailable, process-specific vendor tools, these features can not be efficiently synthesized in a standard-cell process. As researchers implementing an architectural idea, we found it easier (and more cost effective) to alter the design by adding memory interleaving and access arbitration.

Achieving a fast clock is worth the research effort: The RTL model we describe in Section 4 is *not* version 1. The PE has

[3] The estimate is "rough", because scaling two lithography generations to produce an area estimate is far more problematic than a single generation.

been through 4 major and an uncounted number of minor revisions. Each of the major revisions was caused by an inability to meet timing. The first version, which was a fairly straightforward translation of the originally described architecture [1] was slow, achieving 80-90 FO4. Reducing that to 65 FO4 (version 2) was difficult, but required only small changes to the design (fewer ports on memory structures, less wiring). Achieving 30-40 FO4 (version 3) took deeper pipelining, which changed the entire microarchitecture. The 20.3 FO4 in the PE in this paper (version 4) required both microarchitectural changes (the addition of speculation to the instruction scheduler being the most significant), along with tens of small redesigns (most of which pushed logic from one pipeline stage to another and added fast paths for common-case execution situations). Without the goal of a high clock rate, many of these microarchitectural innovations would not have been attempted.

Small wires are not costly: With all of the recent architectural concern for wire-delay on long distance communication paths, a different phenomenon has occurred for short wire segments. There are now so many metal layers that one pays very little area overhead for local, but fairly wide and complex crossbars. Cadence Encounter is able to route and add clocking and buffers to the WaveCache's 8 PE domain with only a 8.6% expansion in the PE cell area. The reason is that the interconnect sits above the rest of the design, with the intra- and inter-cluster interconnects higher up still. Note that, despite the low impact on area, care must still be taken with delay. Traversing just the wire in the domain interconnect requires a full 1/2 clock, plus 3/4 of a clock on each end for processing.

8 Conclusion

This paper has described a pipelined, synthesizable RTL model of the WaveScalar processor. In combination with detailed simulation studies, the model demonstrates that a high-performance WaveCache can be built in current generation technology. A 16-cluster design consumes $252\mathrm{mm}^2$of silicon in 90nm with a cycle time of 25 FO4 (20.3 FO4 for PE execution cores). This design delivers over 50 IPC for multithreaded Splash2 applications and up to 125 IPC for hand-coded dataflow kernels. Larger, more aggressive WaveCache designs that will be feasible in future technology generations can achieve even greater performance without any significant changes to the microarchitecture.

References

[1] 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.

[2] "TCB013GHP - TSMC 0.13um core library." Taiwan Semiconductor Manufacturing Company, Ltd. Release 1.0. October 2003.

[3] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the pentium 4 processor," *Intel Technology Journal*, 2001.

[4] 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.

[5] J. H. Ahn, W. J. Dally, B. Khailany, U. J. Kapasi, and A. Das, "Evaluating the imagine stream architecture," in *Proceedings of the 31st annual international symposium on Computer architecture*, p. 14, IEEE Computer Society, 2004.

[6] D. R. Ditzel, J. L. Hennessy, B. Rudin, A. J. Smith, S. L. Squires, and Z. Zalcstein, "Big science versus little science–do you have to build it? (panel session)," in *Proceedings of the 17th annual international symposium on Computer Architecture*, p. 136, ACM Press, 1990.

[7] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife machine: Architecture and performance," in *Proc. 22nd Annual International Symposium on Computer Architecture*, June 1995.

[8] W. S. Tan, H. Russ, and C. O. Alford, "Gt-ep: a novel high-performance real-time architecture," in *Proceedings of the 18th annual international symposium on Computer architecture*, pp. 13–21, ACM Press, 1991.

[9] Y. Kodama, H. Sakane, M. Sato, H. Yamana, S. Sakai, and Y. Yamaguchi, "The em-x parallel computer: architecture and basic performance," in *Proceedings of the 22nd annual international symposium on Computer architecture*, pp. 14–23, ACM Press, 1995.

[10] T. Higuchi, T. Furuya, K. Handa, N. Takahashi, H. Nishiyama, and A. Kokubu, "Ixm2: a parallel associative processor," in *Proceedings of the 18th annual international symposium on Computer architecture*, pp. 22–31, ACM Press, 1991.

[11] S. Borkar, R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb, "Supporting systolic and memory communication in iwarp," in *Proceedings of the 17th annual international symposium on Computer Architecture*, pp. 70–81, ACM Press, 1990.

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

[13] M. Nakajima, H. Nakano, Y. Nakakura, T. Yoshida, Y. Goi, Y. Nakai, R. Segawa, T. Kishida, and H. Kadota, "Ohmega: a vlsi superscalar processor architecture for numerical applications," in *Proceedings of the 18th annual international symposium on Computer architecture*, pp. 160–168, ACM Press, 1991.

[14] M. D. Noakes, D. A. Wallach, and W. J. Dally, "The j-machine multicomputer: an architectural evaluation," in *Proceedings of the 20th annual international symposium on Computer architecture*, pp. 224–235, ACM Press, 1993.

[15] B. D. de Dinechin, "Stacs: a static control superscalar architecture," in *Proceedings of the 25th annual international symposium on Microarchitecture*, pp. 282–291, IEEE Computer Society Press, 1992.

[16] 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.

[17] 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.

[18] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture," in *Proceedings of the 31st annual international symposium on Computer architecture*, p. 52, IEEE Computer Society, 2004.

[19] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 7, IEEE Computer Society, 2003.

[20] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: a scalable architecture based on single-chip multiprocessing," in *Proceedings of the 27th annual international symposium on Computer architecture*, pp. 282–293, ACM Press, 2000.

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

[22] "International technology roadmap for semiconductors." Semiconductor Industry Association, http://public.itrs.net/, 2003.

[23] "Silicon design chain cooperation enables nanometer chip design." Cadence Whitepaper. http://www.cadence.com/whitepapers/.

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

[25] J. Becker and M. Vorbach, "Architecture, memory and interface technology integration of an industrial/academic configurable system-on-chip (CSoC)," in *Proceedings of the Annual Symposium on VLSI (ISVLSI'03)*, pp. 107–112, IEEE Computer Society, Feb. 2003.

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

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

[28] "Multigranular thread support in WaveScalar." In submission to MICRO 2005.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] P. S. Barth, R. S. Nikhil, and Arvind, "M-structures: Extending a parallel, non-strict, functional languages with state," Tech. Rep. MIT/LCS/TR-327, MIT, 1991.

[37] 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.

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

[39] 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.

[40] C. Scheurich and M. Dubois, "The design of a lockup-free cache for high-performance multiprocessors," in *Supercomputing '88: Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, (Los Alamitos, CA, USA), pp. 352–359, IEEE Computer Society Press, 1988.

[41] 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.