Reducing Control Overhead in Dataflow Architectures

Andrew Petersen
Andrew Putnam
Martha Mercaldi
Andrew Schwerin
Susan Eggers
Steve Swanson
Mark Oskin

Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350
{petersen,mercaldi,swanson,aputnam,schwerin,oskin,eggers}@cs.washington.edu

ABSTRACT

In recent years, computer architects have proposed tiled architectures in response to several emerging problems in processor design, such as design complexity, wire delay, and fabrication reliability. One of these architectures, WaveScalar, uses a dynamic, tagged-token dataflow execution model to simplify the design of the processor tiles and their interconnection network and to achieve good parallel performance. However, using a dataflow execution model reawakens old problems, including the instruction overhead required for control flow. Previous work compiling the functional language Id to the Monsoon Dataflow System found this overhead to be $2-3\times$ that of programs written in C and targeted to a MIPS R3000.

In this paper, we present and analyze three compiler optimizations that significantly reduce control overhead with minimal additional hardware. We begin by describing how to translate imperative code into dataflow assembly and analyze the resulting control overhead. We report a similar $2-4\times$ instruction overhead, which suggests that the execution model, rather than a specific source language or target architecture, is responsible. Then, we present the compiler optimizations, each of which is designed to eliminate a particular type of control overhead, and analyze the extent to which they were able to do so. Finally, we evaluate the effect using all optimizations together has on program performance. Together, the optimizations reduce control overhead by 80% on average, increasing application performance between 21-37%.

Categories and Subject Descriptors: C.1.3 [Processor Architectures]: Data-flow Architectures; D.3.4 [Programming Languages]: Compilers

General Terms: Performance

Keywords: tiled architecture, compiler, dataflow, Wavescalar

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PACT'06, September 16–20, 2006, Seattle, Washington, USA.
Copyright 2006 ACM 1-59593-264-X/06/0009 ...$5.00.

1. INTRODUCTION

To address a set of critical, emerging problems in processor design, including complexity, wire delay and fabrication reliability, many computer architects are shifting 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 and exploiting the parallelism this provides. Examples of these *tiled architectures* include RAW [1], SmartMemories [2], TRIPS [3] and WaveScalar [4]. A simple PE decreases 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 a scalable architecture that enables a chip designer to target different levels of performance with different area budgets [5].

One of these tiled architectures, WaveScalar, is a tagged-token dataflow machine [6, 7, 8, 9, 10, 11, 12]. It leverages two key properties of dataflow computing, explicit instruction-to-instruction communication and the dataflow firing rule, to simplify the microarchitecture [4, 13, 5]. Direct producer-to-consumer instruction communication enables WaveScalar processors to optimize their communication infrastructure around local point-to-point, packet-based routing networks. The dataflow firing rule allows decisions about when an instruction can execute to be made locally, enabling the construction of a distributed processor free of complex control mechanisms.

While using the dataflow execution model eases the complexity of *building* tiled processors in today's process technology, it brings with it an old set of problems related to efficiently executing applications on a dataflow machine: (1) an inability to execute imperative language code [14], (2) the "parallelism explosion" of excessive loop iteration generation [15], and (3) the high control-instruction overhead required for correct control flow [16]. WaveScalar solves the first of these problems with a new dataflow interface to memory called wave-ordered memory [4]. It addresses the second problem with k-loop bounding [12], a well-known solution. The third problem, control-instruction overhead, is the subject of this paper.

Control-instruction overhead consists of the instructions that determine the correct flow of execution within a pro-

WaveScalar Instruction Set (single-threaded)			
Operation	Inputs	Outputs	Description
Data steering			
Select	w:d, w:c	w:d	Pass d to one of two outputs based on c.
MERGE	w:d_1, w:d_2,w:c	w:$c?d_1 : d_2$	Pass d_1 or d_2 depending upon c.
INDIRECT-SEND	w:d, w:a	w:d	Send a message w:d to the instruction at address a.
Tag management			
Wave-Advance	w:d	$w+1$:d	Increment wave tag w.
N-WAVE-ADVANCE	w:d, w:v	$w+v$:d	(signed) add to wave tag.
WAVE-TO-DATA	w:$*$	w:w	Extract wave tag.
DATA-TO-WAVE	w:u, w:d	u:d	Modify wave tag.

Table 1: The WaveScalarISA: In addition to RISC-like instructions (add, load, etc.) and memory instructions, the WaveScalar instruction set contains instructions that facilitate data steering and tag management, shown here. The instructions that are the focus of the optimizations presented in this paper are shown in bold.

gram. On von Neumann processors this is largely comprised of branch instructions, which constitute roughly 15% of dynamic instruction count [17]. In contrast, tagged-token dataflow machines have two types of control instructions: *data-steering instructions* and *tag management instructions*.

Data-steering instructions explicitly guide data values to the correct path after a branch. Each live value requires its own data-steering instruction, leading Hicks et al. to observe on the order of twice as many data-steering instructions as execution instructions in their programs [16]. Our own results validate this prior research, as we also find data-steering instructions to be up to twice as numerous as computation instructions.

Tag management instructions are inserted into tagged-token dataflow programs to differentiate between multiple dynamic instances of named program values (for example, variables in simultaneously executing iterations of a loop). Each of these values is flagged with a distinct tag, enabling their iterations to execute in parallel. WaveScalar uses a purely software-driven approach to tag management. The compiler is responsible for managing the tag space, including updating the tags for all live values. Our data show that tag management instructions occur as frequently as data-steering instructions, up to twice as often as computation instructions.

For WaveScalar to be a viable alternative for future processors, this overhead must be reduced. We present three compiler optimizations that reduce control overhead by 80% to only 30-45% of dynamic instructions. The optimizations exploit the fact that control overhead instructions are, for the most part, easily executed in parallel with computation [12]. The optimizations increase compile times by a negligible amount and require only trivial changes to the microarchitectural implementation.

We begin by providing a brief overview of the WaveScalar architecture and microarchitecture. This overview summarizes material that has been published elsewhere [4, 5]. Next, in Section 3, we illustrate how two simple snippets of C code appear as WaveScalar assembly. These two examples are used throughout the text to illustrate the optimizations. This section also presents an overview of the WaveScalar C compiler and the compiler algorithms used to insert control instructions into an application's dataflow graph.

Section 4 presents two of the three compiler optimizations and their hardware support. We call the optimizations WAVEADVANCE-FOLDING and SELECT-FOLDING, as they conceptually "fold" control operations together with computation instructions. We also evaluate the effectiveness of these optimizations using a simulator of the WaveScalar microarchitecture. Our data show that these two compiler passes reduce control overhead by 67% on average.

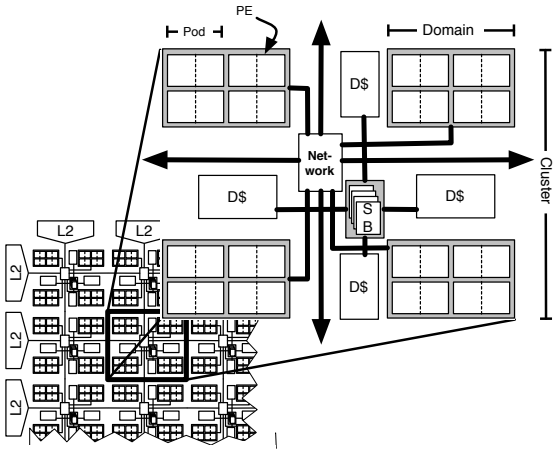

Figure 1: The WaveScalar Processor: The hierarchical organization of the WaveScalar microarchitecture.

We then explore the third control-overhead-reducing compiler optimization, a dataflow graph transformation technique we call FAR-HOISTING. FAR-HOISTING recognizes when live values are unused in loops and minimizes the overhead of updating their tags. Applied alone, this compiler pass reduces control overhead by 22% on average.

We conclude our study by evaluating the effect of applying all three optimizations together. Combining WAVEADVANCE-FOLDING, SELECT-FOLDING and FAR-HOISTING reduces control overhead by 80%, improving application performance by an average of 27%.

2. OVERVIEW OF WAVESCALAR

We begin by presenting WaveScalar, the target architecture for the optimizations presented in later sections. We confine our discussion to a high-level view of the architecture, with specific details included only if they are relevant to the optimizations. A more in-depth description of the architecture appears in [4].

2.1 Dataflow instruction set architecture

WaveScalar is a dataflow architecture. Like all dataflow architectures (e.g. [18, 19, 6, 7, 8, 9, 20, 10, 11, 12]), its binary consists of a program's dataflow graph (DFG). Nodes in the graph are instructions. Arcs are directed and represent operand dependences. While traditional machines

WaveCache Capacity	131,072 static	Clusters	64
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 (min) inter-Cluster: 9 + cluster dist.
PE Output Queue	8 entries, 4 ports (2r, 2w)		
PE Pipeline Depth	5 stages		
L1 Caches	32KB, 4-way set associative, 128B line, 4 accesses per cycle	L2 Cache	16 MB shared, 1024B line, 4-way set associative, 20 cycle access
Main RAM	1000 cycle latency	Network Switch	4-port, bidirectional

Table 2: WaveScalar microarchitectural parameters.

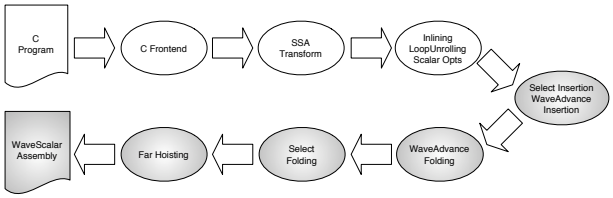

Figure 2: Compiler Flow: The WaveScalar compiler is based on the Scale compiler. Shaded boxes represent compilation passes added to compile and optimize WaveScalar code.

have a program counter that walks a program's control flow graph (CFG) and fetches instructions into a processing core, dataflow machines do not. Instead, they have a "token store", which stores tokens comprised of operand values and their instruction-identifying tags and matches input operands to instructions. The token store in WaveScalar is distributed across multiple processing elements (PEs): each PE contains a small matching table that serves as the token store for the instructions it will execute. When all of the operands for a particular instruction have arrived in this table, the instruction can be executed. This is known as the *dataflow firing rule* [18, 19].

The WaveScalar instruction set comprises RISC-like instructions which carry out most of program execution and special memory operations that preserve memory order for imperative language programs, thread-management instructions for executing multiple threads, and control overhead instructions that facilitate data steering and tag management. Since the semantics of these instructions has been presented elsewhere [4, 13], we will discuss only the control overhead instructions which are the focus of this paper. Table 1 presents a summary.

SELECT takes two input values, a data item and a control bit, and sends the data item to one of two consumers, based on the value of the control bit. SELECT instructions represent 28-42% of all instructions executed in unoptimized WaveScalar binaries. MERGE is the inverse operation of SELECT. It takes two data items and a control bit and, based on the control bit, passes one of the data items to the output. Since the WaveScalar compiler does not yet support software speculation, it does not currently generate MERGEs. INDIRECT-SEND is used for function invocation and accounts for less than 2% of instructions executed. Although we do not address optimizing the use of INDIRECT-SEND in this paper, common optimizations such as function-inlining are effective in reducing them.

In this paper we focus on compiling single-threaded programs, and for these, the key tag manipulation instruction the WaveScalar compiler employs is WAVE-ADVANCE. In WaveScalar, the wave portion of a tag is used to differentiate between several dynamic instances of the same static instructions. For example, different iterations in a loop and different invocations of a function begin on different wave numbers. WAVE-ADVANCE simply increments the wave number. Other wave instructions allow more complicated wave number manipulation, convert wave numbers to operand data, and set the wave field to a specific value.

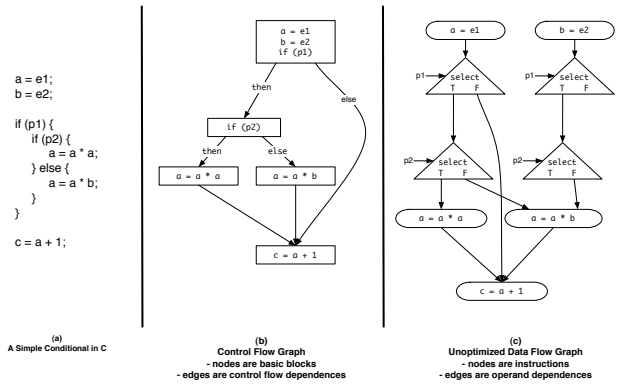

Figure 3: Example of branching code.

2.2 Microarchitecture

Conceptually, each static instruction in a WaveScalar program executes in a separate processing element (PE). Since building a PE for each static instruction is an inefficient use of hardware resources, the WaveScalar compiler maps multiple instructions to a fixed set of PEs, each of which contains a small, local instruction cache. This cache holds up to 64 static instructions at a time. The microarchitecture swaps instructions in and out of these caches as required by program execution.

The microarchitecture consists of a grid of these simple, 5-stage pipelined processing elements. To reduce communication costs within the grid, PEs are organized hierarchically, as depicted in Figure 1. Two PEs are coupled, forming a *pod*; within a pod, instructions can execute and send their results to their partner PE in a single cycle. Four pods are grouped into a *domain*, within which producer-consumer latency is five cycles. Four domains form a *cluster*, which also contains a memory-ordering store buffer and a traditional L1 data cache. A single cluster, combined with an L2 cache and main memory, is sufficient to run any WaveScalar program. To build larger machines, multiple clusters are connected by an on-chip, dynamically routed packet network; communication latency between clusters depends upon the

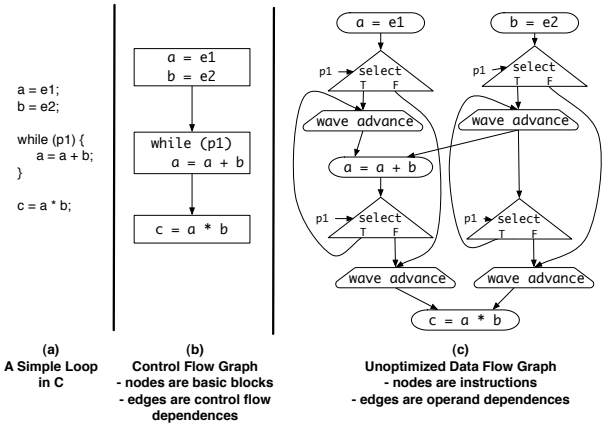

Figure 4: Example of a simple loop.

distance between them on the chip. Data cache coherence is maintained by a directory-based coherence protocol. The coherence directory and the L2 cache are distributed around the edge of the grid of clusters. Table 2 depicts the microarchitectural parameters of the design used in this study.

3. COMPILING FOR WAVESCALAR

Our WaveScalar compiler extends the Scale [21, 22] compiler to support dataflow execution and the WaveScalar ISA. This section presents a brief overview of the compiler, including the ordering of optimization passes. It concludes with two examples of the code produced by the compiler *without* our new control optimizations. Later sections will revisit these examples to illustrate how control optimizations affect the generated code.

3.1 Overview

Figure 2 depicts the overall compilation flow. The unshaded regions represent existing Scale compiler passes, which remain largely unchanged. Our compiler adds the shaded passes to generate and optimize WaveScalar programs.

The WaveScalar compilation path diverges from a conventional compiler at the stage where register allocation would traditionally take place. Being a dataflow machine, WaveScalar has no registers. Instead, an effectively infinite number of pseudo-registers can be stored in the processor as tokens. To exploit this, we base our intermediate representation on single static assignment (SSA) form [23]. Transforming code from SSA form to WaveScalar assembly is straightforward: WaveScalar instructions become nodes in a dataflow graph, and pseudo-register names become arcs that connect these nodes. At this point, only two more compilation passes, SELECT insertion and WAVE-ADVANCE insertion, are required.

Select insertion: SELECT instructions steer data values along the correct control path to consumer instructions. Instead of having an explicit branch instruction, the predicate the branch would normally utilize is computed and directed to a SELECT instruction. Each live-out from every basic block is assigned its own SELECT instruction. The outputs of the SELECT instructions become the inputs to the succeeding blocks.

Tag management insertion: A program's dataflow graph is partitioned into directed acyclic subgraphs or *waves*. For a single-threaded program, tag management insertion consists of inserting a WAVE-ADVANCE on each dataflow arc entering any wave. Inserting WAVE-ADVANCEs in this way ensures that all tokens entering a wave have the same, unique wave number.

WaveScalar does not require a temporal instruction schedule, since execution is driven by the dataflow firing rule and dynamically dispatched by the hardware. However, instructions must be scheduled in space (to particular PEs). Doing so efficiently is the subject of ongoing work [24]. For this paper, we utilize the best instruction scheduler available from that work.

	AIPC	%SELECT	%WAVE-ADVANCE	%other overhead	%total overhead
ammp	0.365	42.4	30.0	7.0	79.4
art	0.812	42.0	34.7	0.7	77.4
equake	1.22	28.8	30.1	3.2	62.1
gzip	0.342	32.8	36.4	2.4	71.6
twolf	0.296	31.4	39.1	5.6	76.1
Mean	0.607	35.5	34.1	3.7	73.3

Table 3: Baseline performance (AIPC) and overhead data: The overhead is calculated as the percentage of dynamic instructions executed that are only for data-steering and only for tag management.

3.2 Examples

In this paper, we will illustrate optimizations with two simple examples. The first is a straightforward *if-then-else* block; the second is a small *while* loop. Figures 3 and 4 depict the C code, associated control flow graph, and unoptimized WaveScalar assembly for these examples.

As Figure 3 demonstrates, at each branch the dataflow assembly must have a SELECT instruction for each live data value to send it to one path or the other at runtime. Similarly, Figure 4 illustrates how live values are routed to each loop iteration. Figure 4 also shows how WAVE-ADVANCE instructions are inserted to uniquely name different iterations of a loop. In the general case, every re-entrant basic block must have a WAVE-ADVANCE instruction for every live input. As discussed earlier, these two forms of control overhead dominate computation.

3.3 Compiler output analysis

To evaluate the quality of the code our compiler generates, we use a cycle-level simulator of the WaveScalar microarchitecture. This simulator closely matches an RTL model [5] for the processor but is significantly faster than RTL-based simulation. The parameters to this simulator are shown in Table 2. For this study, we use five benchmarks from the SPEC2000 benchmark suite: ammp, art, equake, gzip, and twolf. We use this subset because our tool chain can correctly process them.

Table 3 shows the baseline performance data from the unoptimized compiler output. To generate these results, all of the typical compiler optimizations (common subexpression elimination, dead code elimination, etc.) were enabled, but the control-overhead optimization passes we added (de-

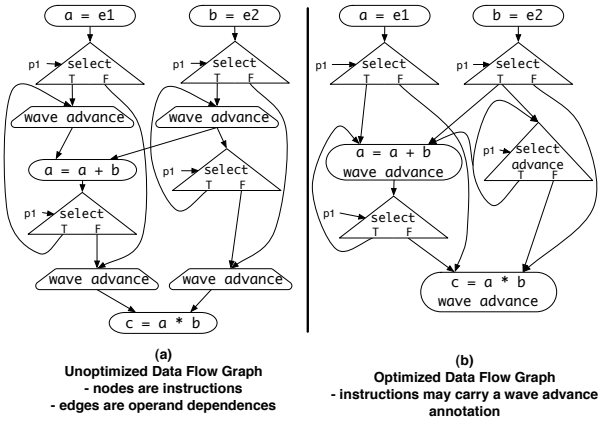

Figure 5: WaveAdvance-Folding: The dataflow graph for an unoptimized simple loop (left) and with WaveAdvance-Folding (right).

scribed below in Sections 4 and 5) were not. Bottomline performance is expressed as AIPC, or Alpha-equivalent instructions per cycle. AIPC only counts computation (non-overhead) instructions completed each cycle to facilitate comparison to a traditional RISC architecture. All control overhead figures (for SELECTs, WAVE-ADVANCEs and total overhead) are calculated as a percentage of the total instructions executed. As evident from Table 3, a significant fraction (70% on average) of dynamic instructions are WAVE-ADVANCE or SELECT instructions. (All other data steering and tag manipulation instructions account for only another 4% of total instructions.)

This data validates prior work which demonstrated that dataflow executes up to 3× more instructions than equivalent imperative code [16]. That work suggested the overhead was partly due to the dataflow execution model and partly due to the non-strict functional language they compiled. Our data show a similar overhead when compiling an imperative language, suggesting that the execution model is entirely responsible for the overhead.

4. ELIMINATING CONTROL OVERHEAD

The previous section showed that the majority of instructions are control overhead. In this section, we introduce two compiler optimizations and a few small modifications to the basic PE design that dramatically lower this overhead. Both optimizations involve merging computation and overhead instructions together into a single instruction, which we refer to as *folding*. The WaveScalar compiler folds WAVE-ADVANCE and SELECT instructions in two distinct passes: WAVEADVANCE-FOLDING and SELECT-FOLDING. Section 4.1 discusses WAVEADVANCE-FOLDING, as the compiler executes this pass first. Section 4.2 covers SELECT-FOLDING. Section 4.3 discusses the changes to the microarchitecture these optimizations require.

4.1 WAVEADVANCE-FOLDING

Algorithm 1 implements WAVEADVANCE-FOLDING, and Figure 5 shows its effect on the example loop code from Figure 4. WAVEADVANCE-FOLDING walks the dataflow graph and folds WAVE-ADVANCE instructions into consumer instructions. Folding produces a composite instruction that

Algorithm 1 WAVEADVANCE-FOLDING: Applied to a list of all instructions in a function

```
1: for all (instr ∈ instructionList) do
2:   for all (producer ∈ instr.producers) do
3:     if (¬IsWaveAdvance(producer)) then
4:       Unfoldable : GotoNext instr
5:     end if
6:   end for
7:   AddWaveAdvanceAnnotation(instr)
8:   for all (waveadvanceInstr ∈ instr.producers) do
9:     BypassWaveAdvance(waveadvanceInstr, instr)
10:  end for
11: end for
```

implements both the original operation of the consumer, as well as an increment of the output value's wave field. Its inputs arrive directly from the WAVE-ADVANCEs' producers rather than through the WAVE-ADVANCE instructions.

Algorithm 2 SELECT-FOLDING: Applied to a list of all SELECT instructions in a function

```
1: for all (select ∈ selectList) do
2:   if (HasSingleProducer(select)) then
3:     target = select.producers
4:     if (IsSelectInstruction(target)) then
5:       combinedOutputs =
             CombineOutputs(select, target)
6:       if (combinedOutputs.numberOfSets ≤ 2) then
7:         predicate = NewPredicate(combinedOutputs)
8:         trueOutputs = combinedOutputs.outputSet1
9:         falseOutputs = combinedOutputs.outputSet2
10:      else
11:        Unfoldable : GotoNext select
12:      end if
13:    else
14:      predicate = select.predicate
15:      trueOutputs =
             select.trueOutputs ⋃ target.outputs
16:      falseOutputs =
             select.falseOutputs ⋃ target.outputs
17:    end if
18:    target.AddSelectAnnotation(predicate,
           trueOutputs, falseOutputs)
19:    RemoveInstruction(select)
20:  end if
21: end for
```

In Algorithm 1, the outermost "for all" loop walks through a list of all instructions in a function. For each *instr* in the list, the algorithm examines all of the *producer* instructions that send inputs to *instr*. If all producers are WAVE-ADVANCEs, *instr* is a legal target for folding. Then, *instr* becomes a composite instruction (line 7), and the sources of its inputs are updated to eliminate the WAVE-ADVANCEs. The loop on lines 8-10 performs the latter step by replacing each input to *instr* from a WAVE-ADVANCE with the inputs to that WAVE-ADVANCE. The bottom node ($c = a * b$) in Figure 5 exemplifies this operation. Originally (left), it receives its inputs from two WAVE-ADVANCE operations. After WAVEADVANCE-FOLDING (right), it receives its inputs directly from four separate SELECT instructions. In this case, two WAVE-ADVANCE instructions were eliminated. The al-

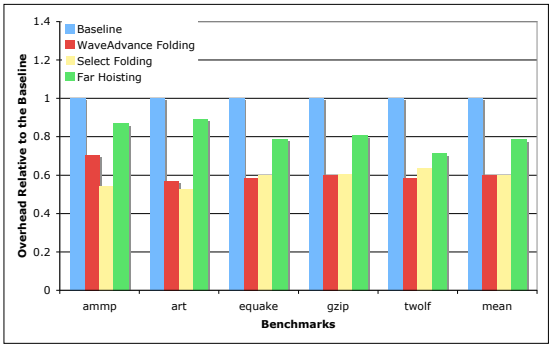

Figure 6: Instruction Overhead: The effect on control overhead of individually applying WaveAdvance-Folding, Select-Folding, and Far-Hoisting.

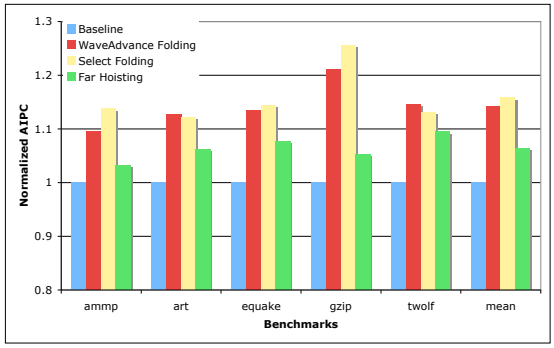

Figure 7: Performance: The effect on baseline performance of individually applying WaveAdvance-Folding, Select-Folding, and Far-Hoisting.

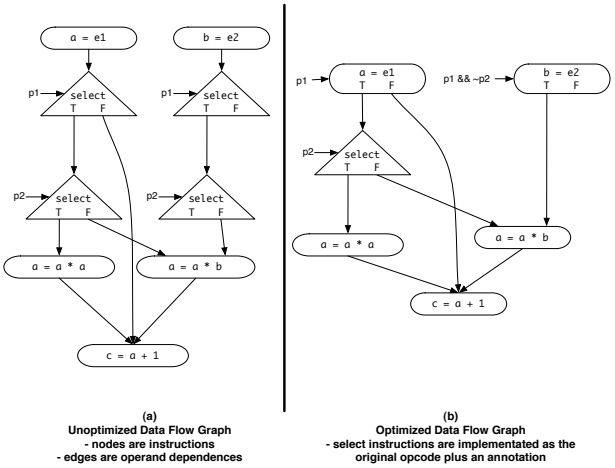

Figure 8: Select-Folding: The dataflow graph for an unoptimized nested branch (left) and with Select-Folding (right). For value 'a,' only one of the Select instructions folds, since folding the second would result in three destinations. For value 'b,' both Selects can be folded, since the predicates can be combined and the resulting instruction has only two destinations.

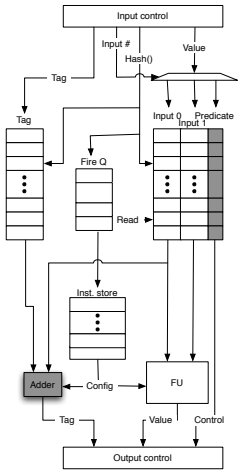

Figure 9: A WaveScalar Processing Element: The shaded elements – a third input queue and an adder – are added to each PE to enable WaveAdvance-Folding and Select-Folding, resulting in an increase in die area of only 2%.

gorithm is not always this efficient. For example, folding the SELECT in Figure 5 bypasses only a single WAVE-ADVANCE.

Note that while the WAVE-ADVANCE instructions have disappeared from the right side of Figure 5, Algorithm 1 never removed a WAVE-ADVANCE instruction. Instead, we rely on a subsequent dead code elimination pass to remove unused WAVE-ADVANCE instructions. Line 9 in the algorithm removes the link from a WAVE-ADVANCE to its former consumer. If all such links are folded away, the WAVE-ADVANCE becomes unused and will be deleted.

Figures 6 and 7 show the results of executing WAVEADVANCE-FOLDING on our benchmark suite. The data show a decline in overhead instructions of 30-45%, which results in an average increase in AIPC of 13% over the baseline performance. AIPC rose because many WAVE-ADVANCE instructions lie on the critical path of execution. (By definition, at least one WAVE-ADVANCE per subgraph in the dataflow graph is on the critical path.) However, the increase in AIPC is not commensurate with the number of instructions removed, since most WAVE-ADVANCE instructions execute in parallel with other, independent instructions.

4.2 SELECT-FOLDING

Algorithm 2 describes the steps involved in SELECT-FOLDING, and Figure 8 illustrates the effect of applying the algorithm to the simple branch example in Figure 3. The SELECT-FOLDING algorithm must solve two high-level problems. First, it identifies when it is legal to fold a SELECT. Second, if a SELECT may be folded, it produces the composite instruction's third input (the predicate) and identifies its outputs.

In Algorithm 2, the outermost loop iterates through a list of SELECT instructions. To determine if *select* may be folded, the compiler determines how many and what type of instructions produce the input value to *select*. If more than one instruction produces the input (for example, the input can arrive from either branch path), then *select* may not

be folded (line 2). In addition, if the producer (*target*) is itself a SELECT instruction and combining the two SELECT instructions would produce an instruction with more than two outputs (checked in lines 4 and 6), *select* may also not be folded. The latter case occurs with the leftmost two SELECT instructions on the left side of Figure 8, which cannot be folded together. The right two SELECT instructions on the left side of that figure may be folded together, however, since combining them produces only a single output.

After determining that *select* may be folded into *target*, the second problem is creating the *predicate* and outputs for the folded instruction. If *target* is not a SELECT instruction, then *predicate* is simply the predicate to the original *select*, and the composite instruction's outputs are the union of *select*'s and *target*'s outputs (lines 14-16). If *target* is a SELECT instruction, however, a new *predicate* must be computed by combining the predicates for both *select* and *target* (see "p1 && ˜ p2" in the example) (lines 7-9). As an example, consider the rightmost composite instruction in Figure 8. This instruction is the result of two iterations of the outside loop in the SELECT-FOLDING algorithm. In the first pass, the node "$b = e2$" is the *target* for the SELECT with predicate $p1$. It becomes "$b = e2$-and-SELECT" with $predicate = p1$ and outputs equal to those of the original SELECT. In the second pass, the new composite instruction is the *target* for the SELECT with predicate $p2$. The two may be folded, because combining their predicates yields only a single output (to the node $a = a * b$).

Earlier, we mentioned that a SELECT cannot be folded if its inputs have more than one producer instruction. In this respect, SELECT-FOLDING differs from WAVEADVANCE-FOLDING, as WAVE-ADVANCEs can be folded no matter how many producers or consumers they have. However, in SELECT-FOLDING, an additional input (the predicate) must be added to the newly annotated instruction. The additional input would have to be sent to *exactly* the producer instruction that is used on that dynamic instance of the SELECT execution. Sending it to the wrong producer would lead to incorrect program behavior. For this reason, it is simpler to avoid this problem by not folding SELECT instructions with multiple producers.

Unlike WAVEADVANCE-FOLDING, SELECT-FOLDING folds SELECT instructions into producer, rather than consumer, instructions. The reason for this lies with the semantics of a SELECT instruction. SELECT takes two input operand tokens: one for a data value and one for the predicate. It sends the data token to one of two consumer instructions, based upon the value of the predicate. If SELECT instructions were folded into consumer instructions, the hardware would need to identify the unused consumer instructions and their tokens (the instructions on the wrong-path) and garbage collect them. Having no other reason to have this facility, adding it to the WaveScalar microarchitecture only for this purpose is a poor implementation choice.

In contrast, folding SELECT instructions into a producer instruction requires no significant changes to the microarchitecture. A third input is added to each PE (described below), and a PE simply matches on three input operands. The third operand is *always* used for SELECT instructions, whether or not they are folded into other instructions.

Figures 6 and 7 show the results of executing SELECT-FOLDING on our benchmark suite. On average, SELECT-FOLDING successfully removes 90% of SELECT instructions (data not shown), which reduces the total control overhead by 40%. The effect of the lower control overhead on performance depends on whether having fewer SELECTs on the critical path outweighs the potential delay of waiting for three, rather than two, input operands to arrive at the composite instruction. In our workload, the combined effect of these factors increased application performance by 15% over the baseline.

4.3 Microarchitectural support

Implementing WAVEADVANCE-FOLDING and SELECT-FOLDING necessitates few changes to the WaveScalar microarchitecture. WAVEADVANCE-FOLDING requires an extra adder in the execute stage of each PE to update the tag field of an output token, while SELECT-FOLDING needs an extra bit in the input operand queues to guide route selection. Figure 9 illustrates such a PE design. The additional hardware results in only a 2% increase in the required die area and no increase in the cycle time.

In addition to WAVEADVANCE-FOLDING and SELECT-FOLDING, this PE can also support the simultaneous execution of computation, data steering, *and* tag management functions, such as ADD-WITH-WAVEADVANCE-AND-SELECT.

Algorithm 3 FAR-HOISTING: Applied to a list of loops in a function

```
 1: for all (loop ∈ loopList) do
 2:   hoistList = ()
 3:   for all (input ∈ loop.inputs) do
 4:     if (¬(input.isUsedIn(loop)∨
              input.isDefinedIn(loop))) then
 5:       hoistList.add(input)
 6:     end if
 7:   end for
 8:   if (hoistList.size ≥ 2) then
 9:     rtoken = AddRendezvousTokenInstructions(loop)
10:     for all (input ∈ HoistList) do
11:       AddDataToWave(rtoken, input)
12:       RemoveHoistedInstructions(input)
13:     end for
14:   end if
15: end for
```

5. FAR HOISTING

Combining WAVEADVANCE-FOLDING and SELECT-FOLDING leads to a 66% reduction in overhead. However, to further reduce overhead, we present a graph transformation optimization that removes values from loops that do not use them. Even if a value is *not* being used in a loop, it must be "spun" through a WAVE-ADVANCE and a SELECT and back to the WAVE-ADVANCE on each iteration. This is done to keep the tag field of the token updated, so it will have the correct wave number when it is eventually used. With one value, this spinning process is not an onerous burden, but with several unused live values in a loop, the overhead quickly adds up.

One solution to this problem is to send the tokens around the loop body entirely and then to "patch" the tag field with the proper tag value. This process is illustrated in the shaded box in Figure 10. To patch the tag values, both the original and final tag value must be known; to do this, the original tag value must be spun through the loop. A

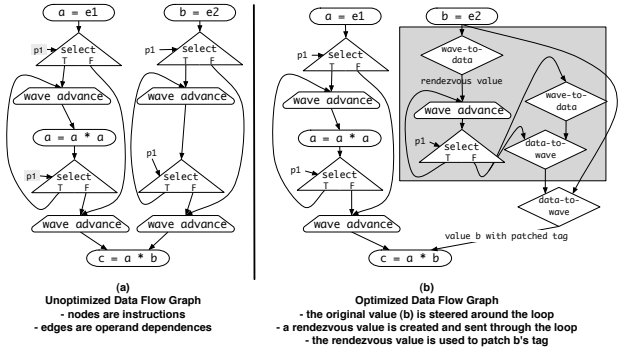

Figure 10: Far-Hoisting: The dataflow graph for an unoptimized loop (left) and with hoisting (right). In this example, a few extra instructions are required, but if enough values are hoisted out of the loop, the overhead required to patch the hoisted values' tags is more than offset by the reduction in Select and Wave-Advance instructions.

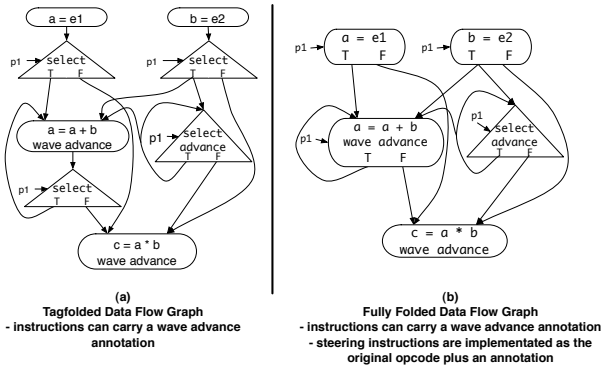

Figure 11: Combining WaveAdvance-Folding and Select-Folding: The dataflow graph for a simple loop that has been WaveAdvance-Folded (left) and then Select-Folded (right). The rightmost Select-Wave-Advance cannot be completely eliminated.

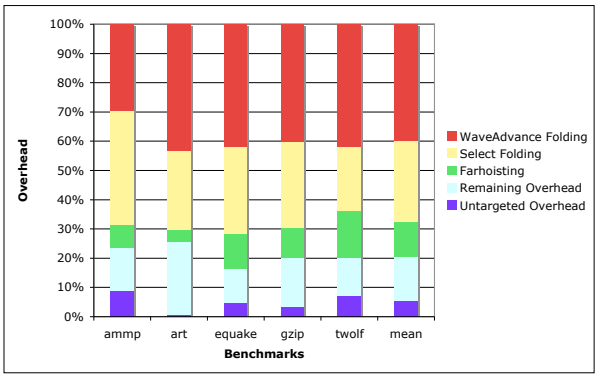

Figure 12: The combined effect of eliminating instructions through WaveAdvance-Folding, Select-Folding, and Far-Hoisting on control instruction overhead. All sources of overhead, both eliminated and remaining, are a percentage of the original control overhead.

Wave-To-Data instruction performs this task; it creates an output token with a data value equal to the tag value of its input token. At the completion of the loop, the tag and value fields of this token are reversed (using Data-To-Wave and Wave-To-Data), so the token will match with tokens that did *not* spin through the loop. This token is used to patch the tag fields of tokens sent around the loop body (via a Data-To-Wave). We call this process Far-Hoisting, as it fast-forwards wave numbers to their post-loop value.

We have implemented a compiler optimization, depicted in Algorithm 3 that performs Far-Hoisting. It works by scanning a function with def-use information. At each loop, it uses the def-use data to determine which values should be hoisted out of the loop, inserts code to produce a rendezvous wave number value, and replaces the associated Wave-Advance and Select instructions of values being hoisted with Wave-To-Data instructions that patch their tags.

In Algorithm 3, the outermost loop iterates through a list of loops in a function. For each *loop*, the algorithm generates a *hoistList* that contains all *input*s to the *loop* that are neither used nor defined within the *loop*. By line 8, *hoistList* is a complete list of all *input*s that may safely be hoisted out of the *loop*. For example, for the left side of Figure 10, *hoistList* would contain the value "*b*". Next, the algorithm creates a rendezvous token (*rtoken*) that is spun through the loop. The resulting code (created on line 9) lies within the shaded box in Figure 10. Finally, for each *input* in *hoistList*, the algorithm replaces the code that spun *input* through the *loop* with a Data-To-Wave that uses *rtoken* to patch the *input*'s tag.

Line 8 in Algorithm 3 specifies when it is potentially profitable to perform Far-Hoisting. The example in Figure 10 is guaranteed to be unprofitable, since only one value (*b*) is being hoisted out of the loop, and it is replaced by an *rtoken* that must be spun through the loop, plus additional instructions to patch *b*'s tag. Therefore, Far-Hoisting is only performed on a *loop*, if *hoistList* contains at least two elements.

Figures 6 and 7 depict the results from applying Far-Hoisting. As detailed in the figures, Far-Hoisting reduces the control overhead 10-20%. Performance increases slightly, because the processor has to perform fewer operations, decreasing contention for its hardware resources (PEs, network links, etc).

6. COMBINING ALL OPTIMIZATIONS

We conclude by examining how the optimizations described in Sections 4 and 5 combine to reduce overhead and improve performance. Figure 11 depicts how our example loop code is transformed when first WaveAdvance-Folding and then Select-Folding are performed, and Figure 12 presents the results.

The three optimizations reduce the original overhead 80% to only 30-45% of total instructions. A small percentage of instructions (8-29% of total Wave-Advance and 8-14% of Select instructions) are *not* folded out of execution. Figure 11 illustrates one of these cases: a Select-and-Wave-Advance remains in the code because it has two producers (the assignment and itself), which violates a Select-Folding rule.

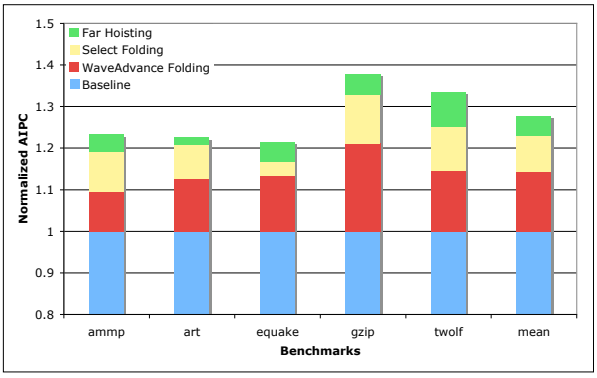

Figure 13: The combined effect of WaveAdvance-Folding, Select-Folding, and Far-Hoisting on normalized AIPC.

The 80% reduction in overhead increases performance 22-37%. As previously discussed, performance does not increase as dramatically as the reduction in overhead might suggest, because overhead instructions are often executed in parallel with the main computation. Nevertheless, the performance improvements show that folding away control instructions and FAR-HOISTING shorten the critical path and reduce contention for WaveScalar processor resources, such as processing elements and network switches.

Folding and FAR-HOISTING have a minimal effect on code size. Individual instruction width increases slightly, since both WAVEADVANCE-FOLDING and SELECT-FOLDING require an additional bit and SELECT-FOLDING potentially increases the number of output operand destinations for the folded instruction. (Additional inputs do not increase code size, since input addresses are not encoded.) However, total code size decreases by 4% on average due to the reduced number of instructions.

7. RELATED WORK

Compiling to the WaveScalar processor builds upon previous work on compilation for dataflow machines [16, 25, 26, 12, 27, 21, 22, 28, 29]. Several of these researchers investigated the effects of control overhead on dataflow execution [16, 25, 26, 30] and using hardware to improve execution performance [31, 15, 32].

Traub, Arvind, et al. compiled Id to the MIT Tagged-Token Dataflow Architecture (TTDA) [12, 27]. The TTDA uses instructions to handle both data steering and tag management, and its PE computes values and tags in parallel. However, TTDA tags are more complex than WaveScalar tags and must be recomputed at every instruction. They do not attempt to merge explicit tag management instructions into other instructions, nor do they attempt to reduce data steering overhead.

Arvind, Hicks, et al. compiled Id to Monsoon [16, 25]. They reported a code-size and cycle overhead of $2-3\times$ over a MIPS-R3000 running C and classified several sources of inefficiency that were created when compiling a "non-strict" language like Id to Monsoon, including the language runtime-system, control instructions, and tag management. After extensive work optimizing their code, the Monsoon team concluded that the instruction-level parallelism obtained by explicitly controlling fanout and joins is better controlled by hardware than software. In this paper, we confirm that the sources of overhead they discovered when compiling a functional language are also present when compiling an imperative language. This suggests that it is the dataflow execution model, rather than any language model, that is the source of the overhead. Furthermore, we develop three optimizations that enable us to handle control in software, while eliminating much of the instruction overhead.

Nikhil, et al. targeted Id to the abstract P-Risc machine and more traditional control-flow computers [26, 33]. They also found the number of control flow operations (forks and joins) to be an issue and developed optimizations to remove them. Their fork and join instructions are different from WaveScalar's, since the P-Risc machine assumes the hardware has resources for storing values and only uses the fork and join instructions for synchronization. Furthermore, overhead instructions are removed, rather than combined with other instructions.

Culler, Goldstein, et al. compiled Id to the CM-5 [34] via the Threaded Abstract Machine (TAM) [35, 30]. They reported that only 34% of instructions executed on their benchmarks were control instructions, because TAM uses local storage and an instruction pointer to avoid steering data values to consumers.

Budiu, et al [32, 36, 29, 28] compiled C to application-specific hardware (ASH). They recently used ASH to compare the performance of static dataflow machines and modern superscalars and then discussed hardware structures commonly found in superscalars that give them a performance edge. Previously, they presented optimizations that decrease the number and latency of memory accesses performed in their system. In particular, the goal of their loop invariant code motion is quite similar to ours for FAR-HOISTING. However, the spatial computing hardware they use eliminates the need to update tags, which simplifies loop invariant code motion and naturally eliminates tag management overhead. Data steering is handled with multiplexors, though they implement several passes to remove unnecessary muxes and combine others.

Finally, the WaveScalar compiler is based on the Scale compiler [21, 22], which targets the TRIPS architecture. TRIPS executes frames, each of which consists of a statically defined group of instructions. Groups are formed by merging basic blocks; this process duplicates some code but reduces the number of branches by merging predicates. TRIPS does not suffer from high control overhead, as it is a von Neumann machine, with a program counter, register file, etc.

8. CONCLUSIONS

WaveScalar is a processor designed to confront the challenges of wire-delay, design complexity, and scalable performance. Being a dataflow processor enables it to achieve these goals; however, it must also address problems inherent to this model of execution. One of those challenges, which has been known for over a decade, is the high cost of control in dataflow models. Prior research found this overhead to be about $3\times$, and we found a similar cost.

This paper presents a solution composed of two pieces, a compiler and a microarchitecture, that work together to significantly reduce control overhead. The basis of the solution lies in recognizing that overhead instructions are relatively simple to implement in hardware and can generally execute

in parallel with computation. This means the microarchitecture can be tuned to execute overhead instructions in parallel with computation instructions.

The compiler must be modified to take advantage of this microarchitecture, and this paper presents two algorithms, WaveAdvance-Folding and Select-Folding, that do so. It also presents a third algorithm, Far-Hoisting, which transforms a dataflow graph to remove unnecessary control instructions by removing unused values from loops. Ultimately, these three algorithms and the microarchitecture work together to reduce overhead to only 30-45% of executed instructions and improve performance by 27%.

9. REFERENCES

[1] M. Taylor, J. Kim, J. Miller, D. Wentzla, F. Ghodrat, B. Greenwald, H. Ho, m Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: A computational fabric for software circuits and general purpose programs," *IEEE Micro*, vol. 22, pp. 25–35, 2002.

[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 *the 34th International Symposium on Microarchitecture*, 2001.

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

[5] S. Swanson, A. Putnam, M. Mercaldi, K. Michelson, A. Petersen, A. Schwerin, M. Oskin, and S. Eggers, "Area-performance trade-offs in tiled dataflow architectures," in *the International Symposium on Computer Architecture (ISCA)*, 2006.

[6] T. Shimada, K. Hiraki, K. Nishida, and S. Sekiguchi, "Evaluation of a prototype data flow processor of the sigma-1 for scientific computations," in *the 13th International Symposium on Computer Architecture*, pp. 226–234, IEEE Computer Society Press, 1986.

[7] 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.

[8] G. M. Papadopoulos and K. R. Traub, "Multithreading: A revisionist view of dataflow architectures," in *the 18th International Symposium on Computer Architecture*, pp. 342–351, ACM SIGARCH and IEEE Computer Society TCCA, May 27–30, 1991.

[9] S. Sakai, y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba, "An architecture of a dataflow single chip processor," in *the 16th International Symposium on Computer Architecture*, pp. 46–53, ACM Press, 1989.

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

[11] 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 *the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[12] 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.

[13] "The WaveScalar architecture," *In submission to ACM Transactions on Computer Systems (TOCS)*, 2006.

[14] Arvind, "Dataflow: Passing the token," in *Keynote at the International Symposium on Computer Architecture (ISCA)*, 2005.

[15] D. E. Culler, K. E. Schauser, and T. von Eicken, "Two fundamental limits on dataflow multiprocessing," in *the IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism* (M. Cosnard, K. Ebcioğlu, and J.-L. Gaudiot, eds.), pp. 153–164, North-Holland Publishing Company, January 20–22, 1993.

[16] J. Hicks, D. Chiou, B. S. Ang, and Arvind, "Performance studies of Id on the Monsoon Dataflow System," *Journal of Parallel and Distributed Computing*, vol. 18, no. 3, pp. 273–300, 1993.

[17] D. Patterson and J. Hennessy, *Computer Organization & Design*. Morgan Kaufmann, 2 ed., 1998.

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

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

[20] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes, "The epsilon dataflow processor," in *the 16th International Symposium on Computer Architecture*, pp. 36–45, ACM Press, 1989.

[21] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. McKinley, "Compiling for EDGE architectures," in *the International Conference on Code Generation and Optimization*, 2006.

[22] R. Nagarajan, S. Kushwa, D. Burger, K. McKinley, C. Lin, and S. Keckler, "Static placement, dynamic issue (SPDI) scheduling for EDGE architectures," in *the International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[23] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451–490, October 1991.

[24] "Instruction scheduling for a tiled dataflow architecture," in *in submission to the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

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

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

[27] K. R. Traub, "A Compiler For the MIT Tagged-Token Dataflow Architecture," Tech. Rep. MIT/LCS/TR-370, MIT, 1986.

[28] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, "Spatial computation," in *the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[29] M. Budiu and S. C. Goldstein, "Optimizing memory accesses for spatial computation," in *the 1st International ACM/IEEE Symposium on Code Generation and Optimization*, 2003.

[30] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken, "Empirical study of a dataflow language on the CM-5," in *the 2nd Workshop on Dataflow Computing*, pp. 187–210, 1992.

[31] D. E. Culler and Arvind, "Resource requirements of dataflow programs," in *the 15th International Symposium on Computer Architecture*, pp. 141–150, IEEE Computer Society TCCA and ACM SIGARCH, May 30–June 2, 1988.

[32] M. Budiu, P. V. Artigas, and S. C. Goldstein, "Dataflow: A complement to superscalar," in *the IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 177–186, 2005.

[33] R. S. Nikhil, "A multithreaded implementation of Id using P-RISC graphs," in *the 6th International Workshop on Languages and Compilers for Parallel Computing* (U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds.), no. 768 in Lecture Notes in Computer Science, pp. 390–405, Intel Corp. and the Portland Group, Inc., Springer-Verlag, August 12–14, 1993. Published in 1994.

[34] S. L. Johnsson, "The Connection Machine system CM-5," in *the 5th ACM Symposium on Parallel Algorithms and Architectures*, pp. 365–366, SIGACT and SIGARCH, June 30–July 2, 1993.

[35] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. V. Eicken, "TAM – a compiler controlled threaded abstract machine," *Journal of Parallel and Distributed Computing*, vol. 18, pp. 347–370, July 1993.

[36] M. Budiu and S. Goldstein, "Compiling application-specific hardware," in *the 12th International Conference on Field Programmable Logic and Applications*, 2002.