Instruction Scheduling for a Tiled Dataflow Architecture

Martha Mercaldi, Steven Swanson, Andrew Petersen, Andrew Putnam, Andrew Schwerin,
Mark Oskin, Susan J. Eggers

University of Washington
{mercaldi,swanson,petersen,aputnam,schwerin,oskin,eggers}@cs.washington.edu

Abstract

This paper explores hierarchical instruction scheduling for a tiled processor. Our results show that at the top level of the hierarchy, a simple profile-driven algorithm effectively minimizes operand latency. After this schedule has been partitioned into large sections, the bottom-level algorithm must more carefully analyze program structure when producing the final schedule.

Our analysis reveals that at this bottom level, good scheduling depends upon carefully balancing instruction contention for processing elements and operand latency between producer and consumer instructions. We develop a parameterizable instruction scheduler that more effectively optimizes this trade-off. We use this scheduler to determine the contention-latency sweet spot that generates the best instruction schedule for each application. To avoid this application-specific tuning, we also determine the parameters that produce the best performance across all applications. The result is a contention-latency setting that generates instruction schedules for all applications in our workload that come within 17% of the best schedule for each.

Categories and Subject Descriptors C.1.3 [*Processor Architectures*]: Other Architecture Styles—Data-flow Architectures; C.0 [*Computer Systems Organization*]: General—Hardware/software interfaces

General Terms Algorithms,Design,Experimentation,Performance

Keywords dataflow, instruction placement, tiled architectures

1. Introduction

Tiled architectures consist of multiple simple processing elements (PEs) connected by an on-chip interconnect. RAW [39], SmartMemories [23], TRIPS [25] and WaveScalar [36] are all examples. Tiled architectures address several emerging, critical problems in monolithic processor design, including design complexity, wire delay, and fabrication reliability. 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 allows a chip designer to target different levels of performance, with different area budgets [37].

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

To achieve their best performance, tiled architectures require an instruction scheduling algorithm that is adapted to their tiled nature. (Our results comparing several algorithms show over an order of magnitude difference in their performance when executing single-threaded applications.) For monolithic processors, instruction schedulers focus on deciding *when* an instruction should be fetched. For example, scheduling a load instructions early helps hide its latency. On a tiled architecture, the scheduler also decides *where* an instruction will execute [19]. For example, placing dependent instructions on the same or adjacent tiles reduces producer-to-consumer operand latency.

The goal of this work is to find a practical scheduling algorithm that generates efficient code schedules for tiled architectures. Our target architecture is WaveScalar [36]. WaveScalar's microarchitecture is hierarchical. Its basic tile is a *processing element* (PE). Two PEs form a *pod* and share a common low-latency bypass network. Four pods connected via a fixed-latency, pipelined network make up a *domain*. Four domains form a *cluster* and communicate over a fixed-route packet network. Finally, clusters can be replicated, with inter-cluster communication occurring over a dynamically-routed on-chip packet network. We use 16-cluster designs in this study.

On WaveScalar, the scheduling question of *where* to execute each instruction amounts to mapping each instruction to a particular processing element. Because of differences in the microarchitecture above and below the domain level, we broke the mapping process into two phases. In the first phase, called *coarse scheduling*, the scheduler assigns each instruction to a domain. In the subsequent phase, called *fine scheduling*, it assigns each instruction to a specific PE in the domain selected by the coarse phase.

We experiment with three coarse scheduling algorithms: COARSE-BY-FUNCTION, COARSE-BY-TOPOLOGY, and COARSE-BY-EXE-ORDER. COARSE-BY-FUNCTION and COARSE-BY-TOPOLOGY partition instructions according to function and producer-consumer dependencies, respectively. COARSE-BY-EXE-ORDER employs profiling information to partition instructions by their execution order. We also sample three algorithms for fine scheduling: FINE-BUG, FINE-UAS, and FINE-BY-EXE-ORDER. FINE-BUG and FINE-UAS are adaptations of two existing scheduling algorithms, Bottom-Up-Greedy [12] and Unified Assign and Schedule [26], that were developed for clustered microarchitectures. FINE-BY-EXE-ORDER, like COARSE-BY-EXE-ORDER, uses profile information to assign instructions to PEs according to their execution order.

We created placements using each combination of coarse and fine scheduling algorithms. Among the coarse scheduling schemes, COARSE-BY-EXE-ORDER most effectively contained communication within the fewest neighboring domains. This is because its profiling knowledge enables it to place more tightly the subset of a function's instructions that are actually executed. For fine scheduling, FINE-BUG outperformed the others, because it makes the best

trade-off between contention among instructions for PE resources and producer-consumer operand latency.

Based on our analysis of the strengths and weaknesses of the coarse and fine scheduling algorithms, we developed a new fine scheduling algorithm, FINE-DAWG (Depth and Width Graph Scheduling). FINE-DAWG partitions the dataflow graph into subgraphs. Using information about the topology of a dataflow graph, such as the length of producer-consumer chains or the number of consumers of a particular value, it determines the instruction composition of the subgraph. This topological information is parameterized, enabling us to explore the trade-off between and ultimately balance the overheads of operand latency and resource conflicts. With a full exploration of the latency-conflict parameter space, the best FINE-DAWG-generated instruction schedules execute on average 28% faster than those generated by FINE-BUG. This effort is therefore warranted for critical applications. However, if one wishes to dispense with the design-space exploration, using the single FINE-DAWG parameter set that generates the best schedule for all applications still outperforms FINE-BUG by 14%.

In the next section we describe the salient features of the WaveScalar architecture. Section 3 motivates the hierarchical scheduling approach, and describes the coarse and fine algorithms used in this study. In Section 4 we explore how these algorithms perform on the WaveScalar microarchitecture. FINE-DAWG is described and evaluated in Section 5. We survey related work in Section 6 and conclude in Section 7.

2. Overview of WaveScalar

We begin by describing WaveScalar, the target architecture for the scheduling techniques presented here. We confine our description to a high level, except when specific details are relevant to instruction scheduling. A more exhaustive description of the architecture appears in [36].

2.1 Dataflow instruction set architecture

WaveScalar is a dataflow architecture. As with all dataflow architectures (e.g. [9, 8, 17, 33, 15, 28, 31, 14, 27, 7, 3]), an application is represented as a dataflow graph (DFG), with control dependences converted into data dependences. Nodes in the graph are instructions, and directed edges between them represent operand dependences. Unlike traditional processors, dataflow machines do not have a program counter; instead instruction fetch, like instruction execution, is data-driven. Also, instead of a register file, they have a token store which associates tokens, comprised of operand values and instruction-identification tags, to the appropriate instruction. WaveScalar's token store is distributed across the processor, with the processing elements. When all of the operands for a particular instruction have arrived at a PE's token store, the instruction can be executed. This is known as the *dataflow firing rule* [9, 8].

Unlike previous dataflow ISAs, WaveScalar supports a memory model which commits memory accesses in program order. This enables it to execute applications composed in imperative languages, such as C. In WaveScalar, the store buffer implements this memory model, the details of which are not relevant to our work here. The interested reader can find them in [36].

2.2 Microarchitecture

WaveScalar's microarchitecture consists of a grid of simple, 5-stage pipelined dataflow processing elements (PEs). Each static instruction in a WaveScalar program executes in a PE. Each PE contains a small, local instruction cache, which can hold up to 64 static instructions at a time. The microarchitecture swaps instructions in and out of these caches, as program execution requires.

To reduce communication costs within the grid, we organize PEs hierarchically, as depicted in Figure 1. Two PEs are first coupled, forming a *pod*; within a pod, instructions can execute and send their results to their partner PE in a single cycle. Four PE pods are grouped into a *domain*, within which producer-consumer latency is five cycles. Four domains form a *cluster*, which also contains a store buffer and a traditional L1 data cache. A single cluster, combined with an L2 cache and main memory, suffices to run any WaveScalar program. To build larger machines, a dynamically-routed on-chip packet network connects multiple clusters. Communication latency between clusters depends upon how far apart they are on the chip. A directory-based coherence protocol maintains data cache coherence. The coherence directory and the L2 cache are distributed around the edge of the grid of clusters. Table 1 depicts the microarchitectural parameters used for this study.

2.3 Instruction loading

The microarchitecture and the runtime system collaborate to load instructions into the WaveScalar processor. When a token's consumer instruction is not resident in the processor, the processor signals the runtime. The runtime then decides where to place the instruction, either by checking a statically constructed table (produced by the compiler) that maps instructions to PEs, or by using an online algorithm to create a new mapping. It also notifies the microarchitecture of the location of the consumer instruction, to bypass the runtime system for future operands. Eventually, the entire working set of instructions will have been loaded into the PE grid in this manner and the runtime loading mechanism will be largely out of the way of execution.

3. Instruction Scheduling for WaveScalar

Instruction schedules for tiled architectures can have both a temporal (when) and spatial (where) component. The architecture determines the type of schedule required. If there are multiple locations where an instruction can execute, be they ALUs, PEs, more complex tiles or register clusters, then the instruction schedule must have a spatial component. Similarly, any architecture with a program counter will require a temporal schedule. As a dataflow machine, WaveScalar has no program counter. It dispatches instructions dynamically, and so the temporal execution order is determined at runtime. Hence, a WaveScalar instruction schedule consists only of the spatial component.

3.1 Hierarchical Approach

We address instruction scheduling hierarchically in two passes. The first pass, which we call *coarse scheduling* allocates instructions to domains. The second pass, *fine scheduling*, further refines this initial domain assignment by designating instructions to individual processing elements. Figure 2 illustrates the process.

We broke the scheduling problem at the domain level, instead of some other place in the microarchitectural hierarchy, for two reasons. First, the network designs within and between domains are quite different. Within a domain, all communication occurs via a full crossbar interconnect, which has a fixed and relatively short latency. However, between domains communication latency is variable and relatively long, as it traverses a buffered packet-switching network. Second, a domain holds 512 static instructions. This size captures a large enough instruction working-set that a coarse scheduling algorithm can make gross decisions based upon overall program graph structure, without concerning itself about optimizing every intra-domain microarchitectural trade-off.

The hierarchical approach to scheduling also helps to manage the large instruction scheduling problem size. Breaking up the problem into subproblems allows the scheduler to tackle the problem in smaller pieces. WaveScalar has a large number of processing elements (2048 in our target design). With the hierarchical ap-

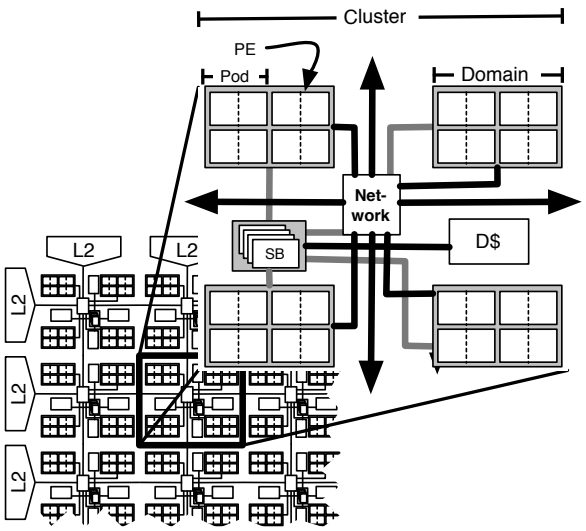

Figure 1. The WaveScalar Processor: The hierarchical organization of the WaveScalar microarchitecture.

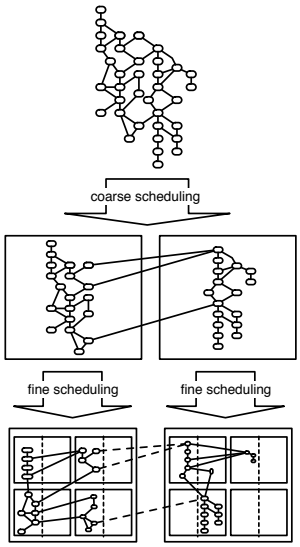

Figure 2. Hierarchical Instruction Scheduling on WaveScalar: The scheduler first assigns each instruction to a domain, and then in a second phase it assigns each instruction to a PE.

PEs per Domain	8 (4 pods)	Domains / Cluster	4
PE Input Queue	16 entries, 4 banks	Network Latency	within Pod: 1 cycle within Domain: 4 cycles within Cluster: 7 cycles inter-Cluster: 7 + cluster distance
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 1. Microarchitectural parameters of the WaveScalar processor

proach, instead of considering all instructions and all processing elements at once, the coarse scheduler need only consider all instructions and the 64 possible domains. Then the fine scheduler is left with some fraction of the instructions (those assigned to a domain) and only 8 possible PE locations for each of those instructions.

In the next two sections we explain the coarse- and fine-grained scheduling algorithms we use in this study.

3.2 Coarse Scheduling Algorithms

The coarse scheduler's job is to partition instructions into large groups and to assign each group to a domain. In this study we examine three approaches. We keep our descriptions of the algorithms to a high level, as the mechanics of applying them to any architecture are simple.

3.2.1 COARSE-BY-FUNCTION

COARSE-BY-FUNCTION allocates all instructions in a function to the same domain. For each function, it cycles through the domains, assigning the function to the first domain that has room for it. If no domain qualifies, it selects the domain which currently contains the fewest instructions, in order to balance the instruction load across domains.

3.2.2 COARSE-BY-TOPOLOGY

Apart from function boundaries, COARSE-BY-FUNCTION does not examine the topology of the application dataflow graph. COARSE-BY-TOPOLOGY inspects the topology in more detail, placing chains of dependent instructions in the same domain. In particular, COARSE-BY-TOPOLOGY performs a depth-first traversal of the dataflow graph, filling domains with instructions in the order in which it encounters them. Thus the algorithm tends to map long chains of producer-consumer instructions to the same domain, thereby localizing instruction communication within a domain.

3.2.3 COARSE-BY-EXE-ORDER

Like COARSE-BY-FUNCTION and COARSE-BY-TOPOLOGY, COARSE-BY-EXE-ORDER fills domains with instructions in sequence, but it does so based on an actual execution profile. As program execution demands each instruction, COARSE-BY-EXE-ORDER assigns it to the current domain. Once this domain is full, COARSE-BY-EXE-ORDER moves on to the next domain. Implementing COARSE-BY-EXE-ORDER requires that the instruction scheduler have access to an execution profile.

3.3 Fine Scheduling Algorithms

The job of a fine scheduling algorithm is to take the output of the coarse scheduling phase, i.e., a mapping of instructions to domains, and to assign each instruction to a specific PE in its domain. Each of the three fine scheduling algorithms described in this section processes domains one at a time.

3.3.1 FINE-BUG

FINE-BUG adapts the Bottom-Up-Greedy (BUG) [12] algorithm, first used in the Bulldog VLIW compiler[11]. BUG was the first phase of a two-phase scheduling strategy. Processing instructions in a bottom-up, breadth-first order, it divided instructions into groups, such that one instruction from each group would form a very long instruction word. The second phase then produced the temporal schedule for each group. Originally, BUG assumed zero communication latency between instructions in different groups. A second version of BUG that appeared in the Multiflow compiler [22] for clustered VLIWs, differentiates between local and remote operand latency. It still processes instructions in a bottom-up, breadth-first order, but attempts to place dependent instructions in the same group, in order to reduce operand latency. This second version of BUG is best-suited to WaveScalar, because it is aware of the non-uniform operand latency within a domain.

Our version of BUG, FINE-BUG, also schedules instructions with a bottom-up, breadth-first traversal of the dataflow graph.[1] However, to place an instruction, FINE-BUG first calculates, for each PE, the number of operands the instruction shares with any of its successors that have already been assigned to the PE. FINE-BUG assigns the instruction to the PE with the largest number of communicating operands. Ties between PEs are broken by round-robin priority.

3.3.2 FINE-UAS

FINE-BUG ignores the resource conflicts that arise when two instructions at the same PE can execute at the same time. When this occurs, one instruction must wait an extra cycle (or more) for the other to complete. Unified Assign and Schedule (UAS) [26] accounts for stalling due to execution conflicts, as well as operand latency. It unifies into a single heuristic both the spatial assignment of instructions to execution locations (to minimize operand latency) and the temporal scheduling of instructions at each location (to minimize execution resource conflicts).

Our implementation of UAS, FINE-UAS, uses a heuristic that reflects WaveScalar's domain resources and latencies. FINE-UAS processes instructions in a top-down, breadth-first order. For each instruction it greedily chooses the best PE, according to a heuristic that estimates when the instruction will execute, based on input operand communication latency and projected resource conflicts.

Because WaveScalar fetches instructions for execution dynamically, it is particularly difficult to determine when two instructions will conflict at execution. We use a simple, but intuitive, heuristic: if instructions are at the same depth in the dataflow graph, we assume they will conflict, otherwise we assume they will execute at different times.

3.3.3 FINE-BY-EXE-ORDER

The third fine scheduling algorithm we consider is a profile-driven scheduler. Like COARSE-BY-EXE-ORDER, FINE-BY-EXE-ORDER uses the dynamic execution order to choose PEs for instructions. As the program executes and instructions are loaded, FINE-BY-EXE-ORDER begins with one PE, filling it to capacity (64 instructions). It then moves on to the other PE in the pod and then to another pod in the domain.

[1] Some implementations of BUG involve two passes through the dataflow graph. The first, bottom-up pass collects "candidate assignments" for each instruction, and then the second pass, top-down, makes the final assignment based on both the locations of the predecessors and candidate locations of the successors. We experimented with this version, but found that the single pass implementation described here was equally effective.

4. Experimental Evaluation

To evaluate the three coarse and three fine scheduling algorithms from Sections 3.2 and 3.3, we produced schedules using each combination of coarse and fine algorithm. We scheduled nine sample applications from the Spec2000 [35] and Splash2 [4] benchmark suites (art, equake, gzip, mcf, radix, twolf and fft, lu, ocean, respectively).[2] The cycle-level simulator used for this study is tuned to match the latencies, resources, and restrictions of an RTL implementation [37] of the architecture.

Table 2 shows the average performance of each of these nine schedules. The IPC for each is normalized to the IPC of COARSE-BY-TOPOLOGY with FINE-BUG. In this section we discuss only the results in the top part of Table 2. We will describe the experiments that produced the last two rows of data in Section 5.

4.1 Coarse Scheduler Evaluation

To evaluate the quality of the coarse scheduling algorithms, we compare the normalized IPCs in each of the first four rows in Table 2. Each row enables us to evaluate: for a given fine scheduling algorithm, how do the coarse schedulers compare? The data show that COARSE-BY-EXE-ORDER has the best overall performance with two of the three fine schedulers.

Two factors account for the better performance. First, COARSE-BY-EXE-ORDER uses execution-order information to pack the subset of static instructions that are actually executed more compactly. (Rarely used paths are placed elsewhere in the processor.) This reduces average operand latency.

Second, COARSE-BY-EXE-ORDER-generated schedules incur cheaper inter-domain operand traffic. The coarse schedulers determine what share of operand traffic crosses domain boundaries. As Figure 3 shows, COARSE-BY-EXE-ORDER and COARSE-BY-FUNCTION each incur approximately the same amount of this traffic (6% and 7%, respectively). However, for COARSE-BY-EXE-ORDER the majority of the inter-domain traffic (85%) is local to the cluster. In contrast, only 14% of COARSE-BY-FUNCTION's inter-domain traffic remains in the cluster, with the rest traversing the more costly inter-cluster routing network. Thus, while each of these two coarse algorithms produces the same proportion of inter-domain traffic, this traffic generally travels farther, with increased latency, using COARSE-BY-FUNCTION.

The one case in which COARSE-BY-EXE-ORDER is not the best strategy is when combined with FINE-BY-EXE-ORDER. In this situation COARSE-BY-FUNCTION produces better performance. Judging only by the data in Figure 3, this is somewhat surprising, because COARSE-BY-FUNCTION incurs the *smallest* amount of intra-domain traffic of the three coarse schedulers. However, when paired with a terrible fine scheduling algorithm (which, as we'll see in the following Section FINE-BY-EXE-ORDER turns out to be), the best coarse strategy is to keep operand traffic *outside* of the domain and out of the hands of the fine scheduler.

4.2 Fine Scheduler Evaluation

To compare the fine schedulers, we examine the values in each column of Table 2. No matter the coarse scheduler, FINE-BUG readily outperforms both FINE-BY-EXE-ORDER and FINE-UAS. We first use simulator-generated data to examine the reasons why

[2] We use these particular applications because both our binary translator-based compiler and WaveScalar simulator can compile and simulate them.

	COARSE-BY-FUNCTION	COARSE-BY-TOPOLOGY	COARSE-BY-EXE-ORDER	Average
FINE-BUG	100%	110%	123%	111%
FINE-UAS	79%	94%	110%	94%
FINE-BY-EXE-ORDER	61%	77%	66%	68%
Average	80%	94%	100%	
FINE-DAWG-MAX	108%	120%	141%	123%
FINE-DAWG-SAME	96%	112%	130%	113%

Table 2. Instruction schedule performance: These IPC values are averages across all applications, and normalized to the performance of COARSE-BY-FUNCTION plus FINE-BUG.

FINE-BY-EXE-ORDER and FINE-UAS fall short, and then analyze FINE-BUG's success.

Based only on the distribution of intra-domain traffic shown in Figure 3, FINE-BY-EXE-ORDER's poor performance is somewhat unexpected. Of the three fine scheduling algorithms, it confines almost all of its intra-domain operand traffic to the cheaper intra-pod communication. The preponderance of very localized traffic resulted in average operand latencies that were 68% lower than that of the other fine schedulers. Operand traffic is not the whole story, however, as execution resource conflicts also contribute to performance. Figure 4 shows that FINE-BY-EXE-ORDER incurs on average 5.3 times the number of ALU conflicts/instruction as the other fine schedulers. Thus, despite near-perfect communication locality, its performance is hampered by its aggressive PE-packing approach to fine scheduling.

As an aside, this validates the intuition described in Section 3 that coarse and fine placement are qualitatively two different problems. The scheduling strategy that produced the best performance at the coarse level, resulted in the worst performance at the fine level. Coarse schedulers can focus single-mindedly on reducing operand latency and capturing instruction set working locality. Fine schedulers, however, must carefully balance latency against resource contention.

FINE-UAS was designed to optimize the operand latency-resource contention trade-off for processors that schedule instructions statically (e.g., VLIW processors). To incorporate the notion of out-of-order execution, we augmented it with a simple heuristic to estimate execution resource conflicts that occur from dynamic dispatch. (Recall that, according to this heuristic, two instructions will conflict if they are at the same depth in the dataflow graph). Our results show that this intuitive and simple heuristic will not account for the variation in instruction order induced by out-of-order execution. Table 2 shows that FINE-UAS performs more poorly than FINE-BUG which does not explicitly consider resource conflicts. Relative to FINE-BUG, FINE-UAS yields not only more execution conflicts (Figure 4), but longer operand latency as well (Figures 3).

The failure of the conflict prediction heuristic can be traced to two factors, both of which relate to the topology of the dataflow graph. First, it is far more likely that two instructions that are "nearby" in the dataflow graph will conflict. This is because they are more likely to execute around the same time, even if they lie at different depths. On the other hand, two instructions in entirely separate subsections, but at the same depth of the graph, are more likely to belong to different temporal phases of execution and therefore will likely not conflct. Second, not all anticipated resource conflicts should carry equal weight. When a dataflow graph is about to fan out, increasing parallelism, it is best to exploit that parallelism by dividing the instructions between multiple PEs; thus, these instructions should be weighted toward separate PEs. However, if potentially conflicting instructions lie on paths which are about to merge together, the scheduler should weigh those conflicts as less likely to slow down execution.

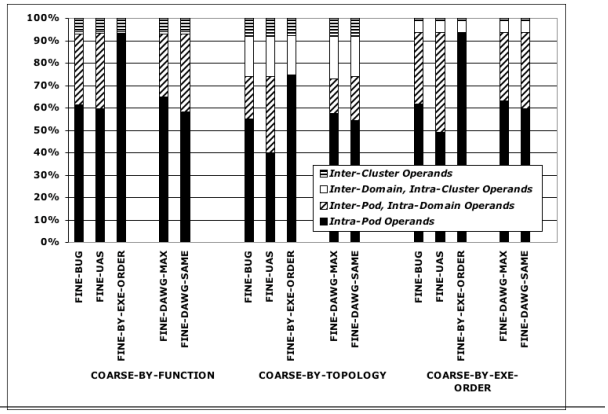

Figure 3. Operand Traffic: A breakdown of the operand traffic incurred by each instruction schedule. This data represents the average across all benchmarks. The coarse scheduling algorithm determines what share of operand traffic travels across domain boundaries, and the fine scheduling algorithm determines how much intra-domain traffic is also intra-PE.

Although more sophisticated heuristics, such as those based on the discussion above, may improve the quality of FINE-UAS schedules, we opted to pursue a more general strategy. Instead, we have developed an algorithm that allows us to explore a range of schedules in the latency-contention trade-off. We will explain this algorithm in Section 5.

FINE-BUG does not use an explicit estimate of operand latency or resource capacity. Nevertheless, its simple approach to balancing these two factors appears successful. The algorithm generally tries to disperse instructions across parallel execution units, except when overridden by operand locality concerns. The resulting middle ground between latency and conflicts is reflected by the data in Figures 3 and 4. Coupled with FINE-BUG's strong IPC results, this indicates that some middle point, trading off communication locality and execution conflicts, is a good design target. This observation motivates the development of FINE-DAWG.

5. FINE-DAWG Scheduling

In this section we describe FINE-DAWG, a new scheduling algorithm designed both to explore the latency-contention design space and to generate quality code for dynamically scheduled processors. FINE-DAWG operates in two phases, first bundling instructions into groups, and then assigning each of these groups to a PE. Before delving into the mechanics of the algorithm, we first describe these two phases informally.

Phase one forms groups based on the topology of the dataflow graph. It takes two parameters, $MaxDepth$ and $MaxWidth$, and carves the dataflow graph into groups of instructions which have dependence chains at most $MaxDepth$ instructions long, with each instruction having at most $MaxWidth$ potentially parallel

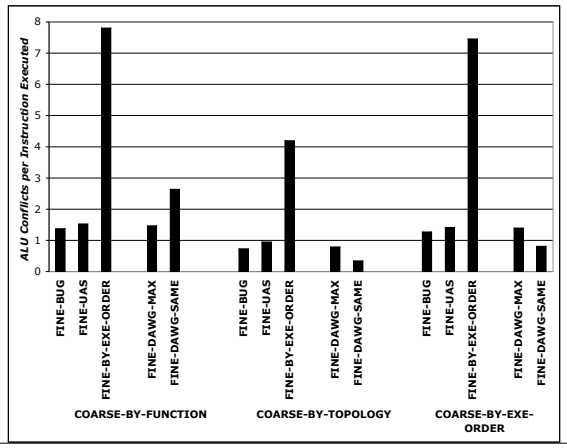

Figure 4. ALU Conflicts: The average number of ALU conflicts per instruction executed for each placement.

successors. The $MaxDepth$ parameter controls how aggressively FINE-DAWG confines operand communication to within a single PE: a high value reduces operand latency, as more dependent instructions are placed at the same PE; alternatively, a low setting increases operand latency by assigning instructions to different PEs. $MaxWidth$ limits the amount of parallelism within a single PE: a high value increases ALU contention, as more parallel instructions are scheduled to the same PE, and a low value has the opposite effect. While all of these values can be selected independently, wise choices will take care to observe the capacity at each PE (64 instructions). Settings where $MaxWidth \times MaxDepth$ vastly exceed this number will likely cause the WaveScalar processor to thrash.

After phase one has assigned all of the instructions in the dataflow graph to groups, phase two maps these groups to PEs using a single parameter, $DepDegree$. $DepDegree$ is a value between zero and one, and determines how aggressively inter-group operand dependencies are used to choose PE locations for groups. A value close to zero raises the probability of dependent groups being placed on the same PE; a value close to one separates them.

We will make this brief description of FINE-DAWG more precise by walking through the algorithm's pseudocode.

Phase One: Phase one gathers instructions into groups based on the topology of the dataflow graph. The algorithm maintains a list of nodes that have no predecessors. To create a group (GRP), FINE-DAWG first calls $CreateGroup(i)$ on an instruction i from this list. Then, after the group is created, it removes the instructions in the group from the graph (adding them to $GROUPED$) and updates the list of predecessor-free nodes. The process repeats until all instructions belong to a group.

Algorithm 1 details how to generate a group, beginning from some instruction i. $CreateGroup(i)$ traverses levels of the dataflow graph iteratively from instruction i, up to $MaxDepth$ levels deep (line 3). At most $MaxWidth$ instructions from each level are added to the group (lines 4 and 5).

Phase Two: Phase two accepts a single tuning parameter, $DepDegree$, and assigns each of the groups produced by phase one ($GROUPS$) to a PE. Initially, all of the PEs contain no instructions. For each group, GRP, FINE-DAWG identifies the two PEs, $pe_{\max}$ and $pe_{\min}$, with whose instructions GRP has most and fewest dependences, respectively. With probability $DepDegree$, it will assign GRP to $pe_{\max}$; otherwise it assigns it to $pe_{\min}$.

Algorithm 1 FINE-DAWG: $CreateGroup(i)$: A portion of Phase One

```
1: GRP = i
2: LEVEL = successors(i)
3: for depth = 1 to MaxDepth do
4:    SELECTED = MaxWidth nodes from LEVEL
5:    GRP = GRP ⋃ SELECTED
6:    GROUPED = GROUPED ⋃ SELECTED
7:    LEVEL = successors(SELECTED)
8: end for
9: return GRP
```

Algorithm 2 FINE-DAWG: Phase Two (applied to the output of Phase One, $GROUPS$)

```
1: for all GRP ∈ GROUPS do
2:    r = rand()
3:    if r < DepDegree then
4:       assign GRP to the PE with which it has the most communication
5:    else
6:       assign GRP to the PE with which it has the least communication
7:    end if
8: end for
```

5.1 Evaluation

We explore the tradeoff between execution resource conflicts and operand latency by exploring the parameter space of FINE-DAWG. Beginning with the domain assignments produced by COARSE-BY-FUNCTION, COARSE-BY-TOPOLOGY, and COARSE-BY-EXE-ORDER, we produced PE assignments using FINE-DAWG, parameterized by each combination of $MaxDepth \in \{2, 4, 8, 12, 16, 32, 50, 64, 128\}$, $MaxWidth \in \{1, 2, 3, 4, 6, 10\}$, and $DepDegree \in \{.1, .5, .9\}$.

We begin our analysis of FINE-DAWG by observing how much of the latency-contention design space it explores. Using COARSE-BY-EXE-ORDER as the coarse scheduler, along with FINE-DAWG with all 162 parameter settings, we scheduled all of the applications in our workload. Figure 5 depicts the average latency-contention for the benchmarks for each parameter configuration. The X-axis measures the percentage of inter-pod operand traffic within total traffic; the Y-axis measures ALU conflicts per executed instruction. The graph also shows three additional labeled axes. These depict how changes in FINE-DAWG-MAX's input parameters effect the resulting schedule output. Finally, we have added points for FINE-BUG, FINE-UAS, FINE-DAWG-SAME, and FINE-DAWG-MAX (the latter two are explained below). (FINE-BY-EXE-ORDER is an outlier, with low latency and extremely high conflicts.)

The data provide several insights about FINE-DAWG. First, it indicates that varying the parameters of FINE-DAWG successfully manipulates both operand latency and ALU conflicts, providing a complete exploration of that trade-off space. Second, it shows that the inter-pod operand traffic threshold below which ALU conflicts rise dramatically falls at 35%. This gives code schedulers the maximal end point for PE instruction occupancy. Lastly, FINE-UAS lies at a non-Pareto optimal point; consequently, because schedules exist that have both lower latency and ALU contention, one would not apply FINE-UAS to WaveScalar.

The charts in Figure 7 show how the performance of FINE-DAWG varies from application to application. The X and Y axes in the chart show the parameter space $MaxWidth$ and $MaxDepth$ respectively for $DepDegree = 0.1$. The shading gradient indi-

cates the performance of each parameter setting relative to the performance for the best parameter setting for that application. The darker the shading, the stronger the performance.

The applications interact with FINE-DAWG's parameters in a variety of ways. In the top row are ocean, mcf, and gzip, for which the vast majority of the parameter settings settings fall in the 90-100% gradient. This consistency is due to "saturation" of the dataflow graph, when forming deeper and deeper subgraphs fails to yield any more improvement in performance. By contrast the applications in the bottom row, fft, radix, and lu, are all extremely sensitive to $MaxWidth$ and $MaxDepth$, and each prefers a slightly different parameter range. This is because kernels drive the performance of these three benchmarks. When the topology of FINE-DAWG's subgraphs fits the kernel well, performance is strong; but if the fit is not good, it quickly drops off. This implies that it may be fruitful to identify and use hotspots to set FINE-DAWG's parameters.

In practice, it is not practical to explore the entire parameter space as we have. As an alternative, we have identified the single parameter set which produced the best overall performance. The analysis in Figure 6 is the basis for this selection. The plot depicts the cumulative distribution functions of the different parameter settings applied to our set of applications. The X-axis is the performance loss relative to the best performing schedule (expressed as a percentage decrease in maximal IPC). For a particular X-value, the Y-axis indicates the number of applications whose performance is at the X-value level or faster. Depending on how close to maximal we require performance to be (the X value), the Y value is an indication of how likely it is that that an application will achieve it.

In order to display the information in Figure 6 more clearly, we have shown only five of the total 162 different parameter combinations used in this study, the two best settings and three others that are representative of the range of results. The data shows that $(MaxDepth, MaxWidth, DepDegree) = (50, 3, .1)$ consistently produced better schedules than all other settings, coming within 17% of maximal for all applications.

Returning to Table 2, we see that the combination of FINE-DAWG with COARSE-BY-EXE-ORDER produces the best performing schedules. We show two data points for FINE-DAWG: FINE-DAWG-SAME and FINE-DAWG-MAX. FINE-DAWG-SAME is the performance when using the strongest consistent parameter set, $(50, 3, .1)$, across all applications. This exceeds the previous best combination of COARSE-BY-EXE-ORDER and FINE-BUG by 14%. FINE-DAWG-MAX is the performance when using the best parameters for each application, which exceeds COARSE-BY-EXE-ORDER and FINE-BUG by 28%.

6. Related Work

Instruction scheduling is a classic compiler optimization problem, which has been widely studied [29, 40, 21, 16]. As they are most closely related to WaveScalar scheduling, we focus our discussion of related work on algorithms designed for tiled architectures which have a spatial component.

Scheduling on Clustered Microarchitectures

Recent processor designs partition hardware resources to counteract increasingly slow communication latencies (relative to computation latencies). Instruction scheduling is used, in part, to minimize the additional cycle(s) spent accessing the remote resources. Several scheduling algorithms attempt to balance operand locality with instruction-level parallelism. We opted to begin our scheduling work by implementing Bottom-Up-Greedy [12], because it is the canonical solution for this type of problem. Unified Assign and Schedule [26] was also chosen, because it is, in essence, a general scheduling framework into which a compiler writer inserts a cus-

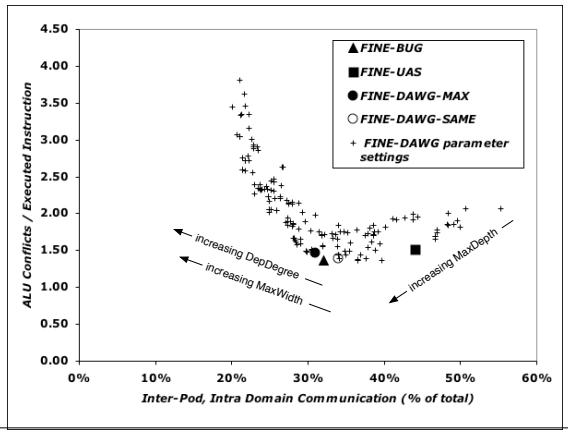

Figure 5. Communication Latency-Resource Conflict Tradeoff Exploration: An illustration of the latency-conflict tradeoff, measured by the percentage of inter-pod operand traffic and ALU conflicts per executed instruction. This data was generated using COARSE-BY-EXE-ORDER as the coarse scheduler and represents the average across all applications. Each cross indicates a design point explored using FINE-DAWG. The plot also shows where in the tradeoff the three previous fine scheduling algorithms, FINE-BUG, FINE-UAS, and FINE-BY-EXE-ORDER, fall.

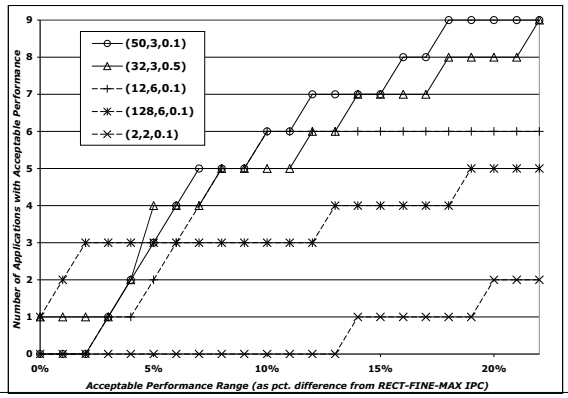

Figure 6. Cumulative Distribution Function of FINE-DAWG Parameters: Each line in this graph corresponds to a different FINE-DAWG parameter setting ($MaxDepth$, $MaxWidth$, $DepDegree$). For ease of display, only five parameter settings of the full 162 are shown: the two best, which have the solid markers, and three from other parts of the spectrum. The X-value defines a range of acceptable schedule performance; each X value is some percentage decrease in IPC from the best performing schedule, called FINE-DAWG-MAX. The Y-axis indicates the number of applications for which a given parameter setting will produce performance at least as high as the value specified by an X value. For example, with the parameter setting (50,3,.1), 5 of 9 applications achieve performance that is no less than 7% lower than the performance of FINE-DAWG-MAX.

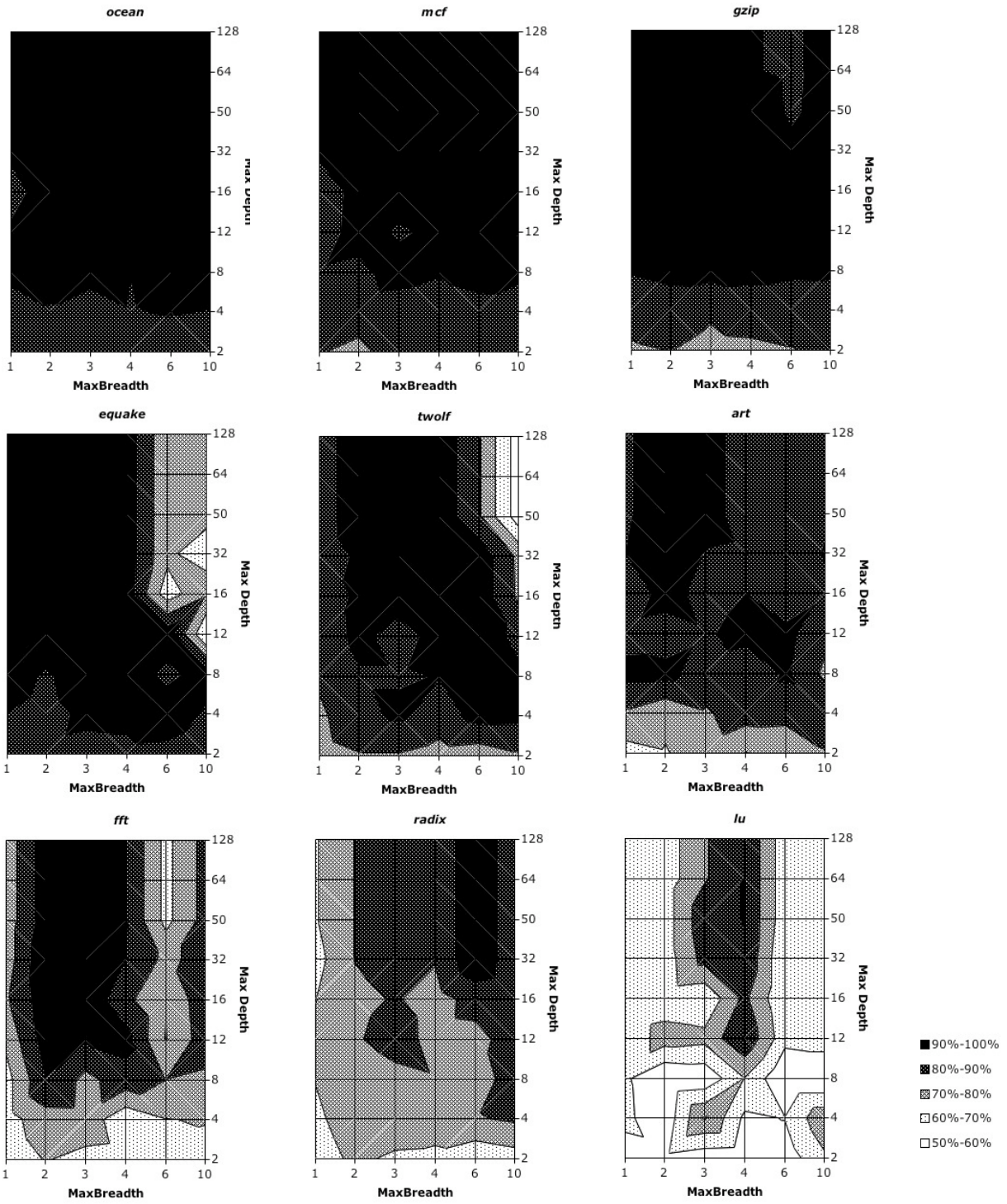

Figure 7. FINE-DAWG Parameter Performance: These plots compare the performance of each parameter setting of FINE-DAWG for each application. The gradient indicates, for each parameter setting, the percentage of the IPC of the maximal parameter setting, it achieves. The behavior of the applications varies dramatically, from extreme parameter insensitivity in the top row to extreme parameter sensitivity in the bottom row.

tom heuristic for the target architecture. The original developers examined several heuristics for clustered microarchitectures [26]. In adapting UAS for WaveScalar, we developed the WaveScalar-specific heuristic described in Section 3.3.2.

Other efforts have focused specifically on clustered VLIW architectures, including modulo scheduling [32], which, like UAS, performs both instruction placement and instruction ordering, and in later versions [43], register spill code insertion, in a single pass. The work by Zalamea et al. [43] studies the effects of program transformations on the success of these scheduling algorithms.

The developers of Lx [13], a clustered VLIW microarchitecture, also studied the spatial aspect of instruction scheduling, and developed another heuristic-based approach to instruction placement [10]. This method employs a pared-down list scheduler to evaluate the ultimate schedule length for a given placement.

Scheduling for Raw

Despite several fundamental differences, Raw [39] and WaveScalar share a grid topology. Raw tiles are arranged in a 2D mesh, with distance-dependant communication latencies between tiles. Raw employs a four-phase instruction scheduler [18, 20]: clustering instructions using dominant sequence clustering [42], merging clusters to match the smaller number of tiles, assigning the clusters to tiles, and finally producing a temporal schedule for the instructions at each tile using a list scheduler.

Scheduling for TRIPS

The TRIPS compiler [24, 6] assigns instructions to locations in its grid of processing elements based on the estimated length of the critical path. The TRIPS compiler applies this algorithm to map each instruction in a hyperblock (up to 128 instructions) on to one of execution tiles. This smaller problem size, coupled with additional mapping constraints such as the location of the register file and memory interface, significantly reduces the space of possible mappings.

Partitioning and Scheduling for Data Cache Locality

Data cache-conscious instruction scheduling has seen two primary thrusts. One approach has been to schedule instructions so as to improve the locality of the data stream for a single processor [34, 5, 30, 41]. The second approach, applied to shared memory multiprocessors, selects and assigns tasks to processors to minimize data sharing between them [38, 1, 2]. Neither of these techniques is particularly well-suited to the scheduling problem that concerns us in this work. The former deals with temporal schedules which do not exist on a dataflow machine. The latter deals with cache coherence issues due to distributed memory; because most WaveScalar execution occurs within cluster boundaries, coherence traffic is small enough that we are comfortable leaving the coherence issue out of placement for the time being.

7. Conclusion

Research is well underway at many universities on tiled architecture designs. This style of microprocessor requires adaptations in the instruction scheduler to achieve peak performance. This paper takes a hierarchical approach to instruction scheduling for a particular tiled architecture, WaveScalar. We find that the hierarchical technique aids in algorithm design, as coarse- and fine-level scheduling are qualitatively different problems. Scheduling at the coarse level, across domains, is entirely about minimizing operand communication latency. However this strategy fails at the fine level, within a domain. There schedulers must carefully balance operand latency with execution resource conflicts. The algorithm developed in this work, FINE-DAWG, explores this trade-off and efficiently identifies the optimal latency-conflict design point. FINE-DAWG also provides a means to attain that goal.

Acknowledgments

This work has been made possible through the generous support of an NSF CAREER Award (ACR-0133188), ITR grant (CCR-0325635), and doctoral fellowship (Swanson); Sloan Research Foundation Award (Oskin); Intel Fellowships (Swanson, Mercaldi); ARCS Fellowships (Putnam, Schwerin); and support from Intel and Dell.

References

[1] S. P. Amarasinghe and M. S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1993.

[2] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1993.

[3] Arvind and R. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3), 1990.

[4] D. Buell et al. *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society, 1996.

[5] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1999.

[6] K. Coons, X. Chen, S. Kushwaha, K. McKinley, and D. Burger. A spatial path scheduling algorithm for EDGE architectures. In *Symposium on Architectural Support for Programming Languages and Operating Systems*, 2006.

[7] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[8] A. L. Davis. The architecure and system method of DDM1: A recursively structured data driven machine. In *Proceedings of the Annual Symposium on Computer Architecture*, Palo Alto, California, April 3–5, 1978. IEEE Computer Society and ACM SIGARCH.

[9] J. B. Dennis. A preliminary architecture for a basic dataflow processor. In *Proceedings of the Symposium on Computer Architecture*, 1975.

[10] G. Desoli. Instruction assignment for clustered VLIW DSP compilers: A new approach. Technical Report HPL-98-13, Hewlett-Packard Laboratories, January 1998.

[11] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. PhD thesis, MIT, 1986.

[12] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. ACM doctoral dissertation award; 1985. The MIT Press, 1986.

[13] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A technology platform for customizable VLIW embedded processing. In *International Symposium on Computer Architecture*, 2000.

[14] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes. The Epsilon dataflow processor. In *Proceedings of the International Symposium on Computer Architecture*, 1989.

[15] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1), 1985.

[16] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1993.

[17] M. Kishi, H. Yasuhara, and Y. Kawamura. DDDP-A distributed data driven processor. In *Proceedings of the International Symposium on Computer Architecture*, 1983.

[18] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a Raw machine. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.

[19] W. Lee et al. Space-time scheduling of instruction-level parallelism on a Raw machine. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating*, 1998.

[20] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *Proceedings of the International Symposium on Microarchitecture*, 2002.

[21] J. L. Lo and S. J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1995.

[22] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. Ruttenberg. The multiflow trace scheduling compiler. *J. Supercomputing*, 1993.

[23] 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.

[24] R. Nagarajan, S. K. Kushwaha, D. Burger, K. S. McKinley, C. Lin, and S. W. Keckler. Static placement, dynamic issue (SPDI) scheduling for EDGE architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[25] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the International Symposium on Microarchitecture*, 2001.

[26] E. Özer, S. Banerjia, and T. M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the International Symposium on Microarchitecture*, 1998.

[27] G. Papadopoulos and D. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the International Symposium on Computer Architecture*, 1990.

[28] G. M. Papadopoulos and K. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *Proceedings of the International Symposium on Computer Architecture*, 1991.

[29] T. A. Proebsting and C. N. Fischer. Linear-time, optimal code scheduling for delayed-load architectures. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1991.

[30] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1998.

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

[32] J. Sanchez and A. Gonzalez. Instruction scheduling for clustered VLIW architectures. In *Proceedings of the International Symposium on System Synthesis*, 2000.

[33] 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 International Symposium on Computer Architecture*, 1986.

[34] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1999.

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

[36] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the International Symposium on Microarchitecture*, 2003.

[37] 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 *Proceedings of the International Symposium on Computer Architecture*, 2006.

[38] R. von Hanxleden and K. Kennedy. Give-n-take - a balanced code placement framework. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1994.

[39] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9), 1997.

[40] K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2000.

[41] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1991.

[42] T. Yang and A. Gerasoulis. PYRROS: static task scheduling and code generation for message passing multiprocessors. In *Proceedings of the International Conference on Supercomputing*, 1992.

[43] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Modulo scheduling with integrated register spilling for clustered vliw architectures. In *Proceedings of International Symposium on Microarchitecture*, 2001.