Instruction Scheduling for Emerging Communication-Exposed Architectures*

Ramadass Nagarajan Doug Burger Kathryn S. McKinley Calvin Lin Stephen W. Keckler

Department of Computer Sciences, University of Texas at Austin

ABSTRACT

Technology trends present new challenges to instruction schedulers and processor architectures. Although increasing transistor counts will enable numerous execution units on a single chip, decreasing wire transmission speeds will cause on-chip latencies to increase to tens of cycles. Conventional architectures, including static VLIWs and dynamic superscalars, and their instruction schedulers are not capable of meeting these challenges. This paper proposes a new instruction scheduling algorithm for emerging wire-dominated architectures that features critical-path instruction selection, placement of instructions to minimize communication distances, and load balancing across the distributed execution units. We evaluate the algorithm on the Grid Processor Architecture (GPA), which supports a hybrid execution model that requires static instruction placement but allows dynamic execution. The combination of this novel scheduling algorithm with the GPA results in the highest demonstrated instruction-level parallelism to date, sustaining over 8 instructions per cycle on a 64-wide issue machine.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*Hardware/ software interfaces*

General Terms

Design, Performance, Algorithms, Measurements

Keywords

instruction scheduling, ILP

1. INTRODUCTION

*This work is supported by DARPA grant F33615-01-C-1892, NSF ITR grant CCR-0085792, NSF CAREER Awards ACI-9984660, CCR-9984336 and CCR-9985109, two IBM University Partnership Awards, two Sloan Foundation Fellowships, and a grant from the Intel Research Council.

Submission to PLDI-2003, June, 2003, San Diego, CA.

The two dominant architectural models, VLIW and dynamic superscalar, take extreme views on the role of static instruction scheduling. The VLIW model relies on the compiler to schedule independent operations to each wide instruction, and it requires guarantees from the compiler that no dependences will be violated. The dynamic superscalar model instead gives the hardware the freedom to execute any instruction on any appropriate ALU, as long as it obeys the original program dependences. Thus, the VLIW model depends entirely on the static scheduler, while the dynamic superscalar pushes much of the complexity of scheduling into the hardware.

While instruction scheduling is well-understood in these existing contexts, the role of instruction schedulers promises to change, as two important technology trends change the future of microprocessor design. First, poor clock scaling will result in wider-issue processors. Second, shrinking devices will result in slow on-chip wires that cause variable multi-cycle on-chip access latencies. Before discussing the impact of these changes on instruction schedulers, we first describe their implications for the superscalar and VLIW models.

The trend towards wider-issue machines, or greater numbers of ALUs, causes problems for both architectural models. Wider-issue is infeasible for superscalar architectures because of the quadratic growth in hardware complexity that occurs when the issue width increases. While VLIW hardware does extend to wider issue, VLIW machines are unable to exploit more ALUs for two reasons. First, the compiler can seldom find enough parallelism to schedule them explicitly because of aliases and other static unknowns, and second, the machine stalls when any of the instructions in the previous VLIW instruction are not complete.

The trend towards larger on-chip latencies also affects the two architectural models. As wires grow smaller with each technology generation, their delay increases. This trend is causing multicycle latencies to appear throughout the processor, such as between ALUs and between ALUs and registers, thus challenging the scalability of each execution model. Clustered VLIW schedulers can model these latencies, but the larger latencies place increased demands on the scheduler, which must try to find additional parallelism to hide them. Wire delays also complicate the task of the superscalar hardware scheduler, and will quickly make it intractable in hardware.

Thus, technology trends are creating a scheduling problem that is too complex for either the compiler or the hardware to solve alone. These trends demand a cooperative approach in which the compiler exposes parallelism and schedules for locality, and the hardware tolerates dynamic latencies and exploits additional parallelism at runtime. While neither the VLIW or dynamic superscalar models support this type of cooperation, there are many possible design

points between these two extremes. In this paper, we evaluate one of these design points and study its impact on instruction scheduling.

This paper presents a scheduling algorithm that uses a critical path node listing, reprioritization after placement, and explicitly models latencies and parallelism between ALUs. We show that with a Static Placement, Dynamic Issue (SPDI) execution model, these scheduling heuristics can greatly reduce the communication-induced critical execution paths among the ALUs. We show that our machine and scheduler improve instructions per cycle (IPC) up to factors of 10 compared with a VLIW and its scheduler, by exploiting both the schedule and runtime adaptability. Our scheduler uses extensible architectural model and evaluation functions that we use to tease apart the effects and interactions of six compiler heuristics. We also evaluate a variety of interconnection topologies and latencies for the SPDI processor. For instance, we show that a tightly packed schedule with more exposed latencies executes much faster than a more loosely packed schedule. We explore the portability of our schedules to fewer numbers of processors to demonstrate the relatively low performance penalty of binary compatibility on this hardware. By finding the right balance between flexibility for the scheduler and complexity in the hardware, we sustain significantly higher instruction level parallelism (ILP) than previously reported in the literature.

The remainder of this paper is organized as follows. Section 2 describes the technology trends that will affect future architecture and scheduler combinations, along with the related work that addresses these emerging challenges. Section 3 describes Grid Processor Architectures, which we use as a case study to evaluate schedulers for future SPDI architectures. Section 4 describes our base top-down greedy scheduling algorithm and the instruction placement optimizations required to reduce distance and latency. Section 5 analyzes the interaction between the scheduler and inter-ALU interconnection network. Section 6 incorporates considerations of different routing latencies and placement constraints into the scheduler. Finally, Section 7 studies the sensitivity of the scheduler's performance to exact knowledge of the routing latencies, and proposes a strategy for generating an abstract schedule that is not bound to a single particular implementation. We summarize and conclude in Section 8.

2. BUILDING AND SCHEDULING SCALABLE ARCHITECTURES

In this section, we discuss the role of static instruction scheduling for future scalable architectures. We first describe trends in architecture. We then describe how out-of-order superscalar processors [29, 15] tolerate imprecision in the scheduler, but that their architectural complexity prevents it from scaling. We then explain how VLIW [7, 25] and partitioned VLIW processors [16, 23] are architecturally scalable, but they rely too heavily on perfection in the scheduler to achieve their promise. Section 3 describes a scalable architecture, which benefits from schedules that exploit parallelism and minimize latencies between ALUs, but tolerates imperfection from the scheduler. We thus hit a sweet spot in the architecture and scheduler co-design space that exploits good static schedules, but mitigates the latencies the compiler cannot predict.

2.1 Technology Trends

Clock Rates: The bulk of microprocessor performance improvements over the past twelve years have come from clock speed increases, at the rate of 40% per year. Clock speeds will continue to increase, but at a slower rate as microprocessors start to hit technology limits [27]. Despite much effort from the research community, the number of instructions executed per cycle on conventional architectures is dropping, not increasing, as a consequence of the faster clock speeds [1, 24]. This drop requires new architectural mechanisms, including multiple ALUs that communicate asynchronously and directly.

Wire Delay: Another consequence of higher clock rates and technology limits is wire delay. In previous microprocessor generations, the processor could reach the entire chip in a fraction of a cycle. However, as wire cross section shrinks, the wires become slower. As a result, future chips will require up to tens of cycles just to route a signal across the chip [1, 12]. To achieve high throughput, the scheduler must consider the latencies between ALUs and shared resources as a new first order constraint. For example, the scheduler should try to place dependent instructions on the same ALU (e.g., 0 cycle latency) or an adjacent ALU (e.g., 1 cycle) since non-adjacent ALUs will incur more latency (e.g., 2 to 20 cycles). The microarchitecture will also need to mitigate the effects of non-uniform delays throughout the processor.

The two current approaches taken to tolerate wire delays are deeper pipelines, as in the Pentium IV, in which two of the twenty pipeline stages are used solely for routing signals along wires, and the clustered Alpha 21264, which maintains two copies of the register file, broadcasting results between them, to reduce register file size and permit partitioning in the processing core. Unfortunately, partitioning the core further causes large drops in instruction-level parallelism(ILP) [2, 3, 6].

2.2 Scalable Architecture Design

To sustain performance improvements, designers must build architectures that achieve high ILP. To motivate our approach, we discuss the problems of scaling the two major classes of ILP machines: dynamic superscalar processors, which issue instructions out of order with respect to the compiler schedule; and VLIW architectures, which obey the strict static schedule dictated by the compiler.

Dynamic Superscalar Processors: Recent papers show that clock speed increases are forcing superscalar processors to their limit, since pipelines can be made only so deep before overall performance starts to drop [1, 24]. Superscalar architectures scale poorly to wide-issue machines because of the quadratic growth (with the issue width) in comparators that check for data dependences, and bypass networks that route results from ALU outputs to consumer instructions. Despite previous predictions in the literature, a 4-wide machine is the widest that has been built to date.

Dynamic superscalar processors select ready instructions and issue them in any order. A wide instruction fetch unit and a large instruction window are in theory all that are needed to exploit available ILP. In practice, the scheduler needs to move high and variable latency instructions as soon as possible in the basic block [14, 19] so that the architecture can use dynamically discovered ILP to hide the latencies. Superscalar processors also need the compiler to produce large basic blocks uninterrupted by control flow, and so far, even with techniques such as unrolling and inlining compilers have not delivered. The typical scheduler for a pipelined architecture uses a greedy approach [10, 22] based on the critical path through the instruction dependence graph (a DAG). It models fixed instruction delays, resource constraints, and other hazards, but typically does not model dynamic events such as cache miss latencies or branch misprediction [26]. For an n-wide machine, the scheduler tries to produce a schedule in which n independent instructions is-

sue every cycle [5]. Balanced scheduling [14] improves over classic scheduling by computing the amount of ILP available in the DAG, and using it to hide the high latency instructions. One advantage of a superscalar architecture is that it can compensate if the compiler does not get the exact order of the instructions right [19].

VLIW: VLIW architectures, conversely, rely on the compiler, not the hardware, to discover and schedule ILP [7, 9, 13, 17, 25]. The compiler guarantees that all instruction placed in a VLIW parallel long instruction word are independent of one another and that the operands are ready to read upon issue. The classic VLIW scheduler also takes a greedy approach [8, 9, 17]. It exploits parallelism by building a ready set where all the instructions in the set can issue in parallel. It then fills the current long instruction word. If there is a choice, it selects the instructions on the critical path first. Software pipelining [17] and similar algorithms focus on loops. It tries to find a steady-state pattern, across loop iterations, in which it fills all of the issue slots of the minimum number of VLIW instructions.

Although the hardware complexity scales linearly with issue width, the problem facing VLIW architectures is that, despite advanced techniques such as predication [20], trace scheduling [8], and treegion formation [11], it is difficult for the compiler to find enough instructions to pack into wide instruction words at compile time. Worse, unpredictable latencies such as cache misses force the entire machine to stall. The hardware thus scales, but VLIW schedulers have proven incapable of finding enough ILP to outperform superscalar processors.

These problems are thus pushing the two hardware/software approaches towards each other, and there are a wide range of possible solutions. We next describe our choice in the middle.

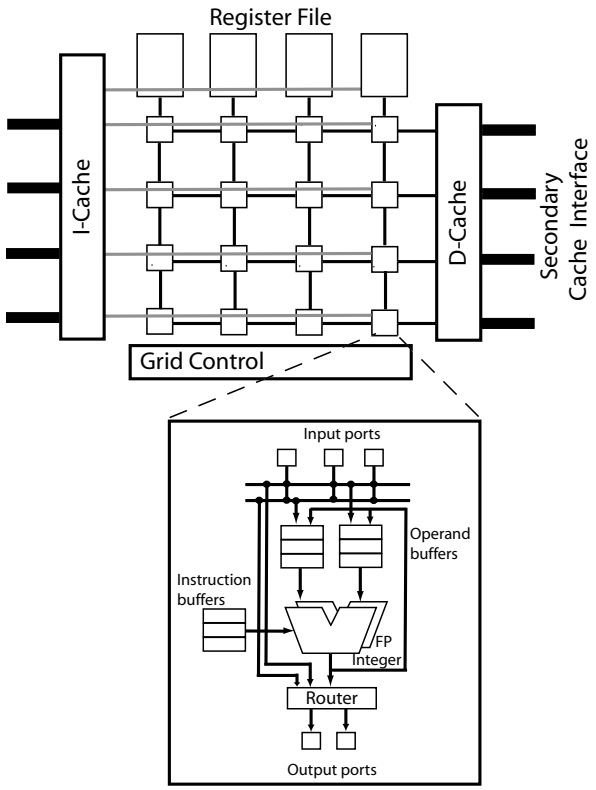


Figure 1: Example 4×4 Grid Processor Architecture.

3. GRID PROCESSOR ARCHITECTURES

The Grid Processor is an emerging architecture that can be considered a hybrid between statically scheduled (VLIW) and dynamically issued (superscalar) architectures [21]. Figure 1 shows the components of a 4×4 grid processor, composed of 16 instruction execution units connected via a thin operand routing network. The instruction cache, register file, and data cache are placed around the perimeter of the ALU array. Each ALU includes an integer unit, a floating point unit, an operand router, and an instruction buffer (storage) for multiple instructions and their operands. Unlike a queue, instructions in these buffers may execute in any order, but an instruction may not execute until all of its operands arrive. While the diagram shows the routing network as a 2-dimensional mesh, the actual topology depends on both hardware and software constraints.

The grid processor is a static placement, dynamic issue processor (SPDI) in which a scheduler statically assigns instructions to ALUs and instruction buffers, but the hardware issues the instructions in dataflow order. The grid processor compiler uses hyperblock generation techniques [20] to create large monolithic blocks of instructions, and then independently schedules each block to the grid. Once a hyperblock has been mapped to the grid, the hardware reads its input registers from the register file and injects them into the grid. Upon arrival at an ALU, these values trigger instructions to fire, which on completion then distribute their results through the operand network to other ALUs. Instructions that produce block outputs write their values back to the register file. The hardware transmits temporary values that are only live within a block directly from producer to consumer, without writing them back to the register file. This strategy helps decouple register allocation from instruction scheduling, but the scheduler may be required to replicate a reused value and schedule the instructions that distribute it to multiple consuming instructions. Address computations for load and store instructions execute within the grid but transmit the addresses (and data values for stores) to the data cache banks. The cache banks send the loaded values back into the grid via the operand network.

Each ALU contains a fixed number of instruction buffer slots. In the grid processor, the corresponding slot across ALUs are collectively called a *frame*. Thus a 4×4 grid processor with 128 instruction buffer entries at each ALU has 128 frames of 16 instruction each. A subset of contiguous frames constitute an *architecture frame*, which is exposed to the compiler for scheduling and placement of a hyperblock. Consequently, dividing 128 frames into 8 architecture frames composed of 16 physical frames would allow the scheduler to map a total of 256 instructions at once to the ALU array. The grid processor hardware uses adjacent architecture frames to speculatively map and execute subsequent hyperblocks concurrently with the non-speculatively executing hyperblock. This technique is very important to good performance, as we illustrate in Section 6. The number of instructions within a single architecture frame represents the size of the instruction window available to the static scheduler. The number of instructions spanning the non-speculative and the speculative frames corresponds to the size of the dynamic scheduling window. In superscalar processors, this window is centralized and relatively small (80-100 instructions), while in the grid processor this window is distributed and can be quite large (thousands of instructions).

The principal issues for the design of the instruction scheduler for a SPDI architecture such as the grid processor include the following:

- **Physical locality:** In the grid processor, the physical distances between the ALUs, the register file, and the cache banks represent latency. Maximizing performance of this

architecture requires the scheduler to place instructions to minimize the communication latencies between dependent instructions and between instructions and the register file and cache banks.

- **Operand network topology:** Maintaining a simple topology and fast routing in the operand network is at odds with a schedule that minimizes routing distance and number of hops in the network. Designing the routing network and static scheduler in concert will balance router speed with connectivity that the scheduler can easily exploit.

- **Fixed frame space:** Since the number of instruction slots (frames) is fixed, the hardware and software must balance frame size with the number of speculative frames. Larger architecture frames may enable a better schedule since the scheduler has more degrees of freedom in placing instructions. However, Section 6 shows that more tightly packed frames increase the number of speculative frames and results in better performance. Thus, the challenge for the scheduler is to create efficient schedules in the smallest number of frames it can.

4. CRITICAL PATH SCHEDULING

In this section, we describe our scheduling algorithm for the Grid Processor. Structurally, our algorithm resembles a greedy VLIW scheduler. It takes as input a group of instructions and a description of the processor model, including communication latencies between different structures. It outputs an assignment of instructions to ALUs. We first describe a simple extension of a VLIW scheduler, which will serve as our baseline GPA scheduler. We then augment the algorithm with several heuristics that improve performance for a given processor configuration.

4.1 Basic GPA Scheduling Algorithm

A classic VLIW scheduler computes the initial root set of ready instructions. It chooses an instruction *i* based on its critical path, and puts the instruction in a VLIW instruction. The scheduler packs as many ready instructions into the current VLIW instruction as it can. After it schedules an instruction *i*, it adds to the ready set any of *i*'s children whose ancestors have already been scheduled. While a VLIW scheduler assigns instruction to an ALU and a time slot, a scheduler in the SPDI model assigns each instruction to an ALU without specifying a time slot. Pictured below is the algorithm of such a scheduler.

```
S = top_down_greedy_sort(hyperblock H);
  foreach instruction i in sorted list S {
    R = find_legal_reservation_stations(i);
    if |R| = 0, Frames++, Reschedule();
    E = sort_reservation_stations(R);
    Slot(i) = prioritize(E);
  }
```

This scheduler first produces an instruction list prioritized by critical path height. We use static instruction latencies and assume no cache misses exist when computing the critical path heights. For the unscheduled instruction *i* with the highest priority, the scheduler computes the set of legal reservation stations *R*. A reservation station *rs* specifies an ALU and one of the slots in the instruction window associated with the ALU. As we will describe in Section 5, the exact interconnection topology defines the set of legal reservation stations. For the most general interconnection framework, the mesh (see Figure 2(d)), all open slots are legal. Other topologies restrict this set to only those reservation stations that are reachable from the locations where the parents of *i* have been scheduled.

If no legal reservation station is available, the scheduler adds to the pool of total reservation stations by increasing the number of frames; it then attempts to reschedule the entire block. If several reservation stations are available, the scheduler chooses the one that is closest to all parents. In particular, the selected slot is one that minimizes the Score:

$$Score(rs) = \max_{\forall p}\{CompleteTime(p) + Distance[rs_p, rs]\}$$

Here p refers to a parent of i, $CompleteTime(p)$ refers to the expected time at which p will produce its results, and $D[rs_p, rs]$ is the number of communication hops required to route p's result to rs. $Score(rs)$ is simply the earliest time at which i can issue at rs. Ties are broken by choosing a slot that is closer to the data caches. We use this algorithm in Section 5 to evaluate the effect of different topological optimizations. We describe a number of additional heuristics to improve our scheduling decisions in the following subsection. We use the mesh topology because it is the most simple, and our results show that it provides the best performance.

4.2 Scheduler Optimizations

We describe three kinds of optimizations that try to balance the twin objectives of maximizing parallelism—scheduling independent instructions on different ALUs—and minimizing communication—scheduling consumers physically close to producers.

- *Locality-Aware Optimizations*: minimize communication latencies along all dataflow paths. In particular, the scheduler attempts to schedule load instructions and dependents of loads closer to data caches. In addition, it attempts to schedule instructions that produce register outputs closer to the register files.

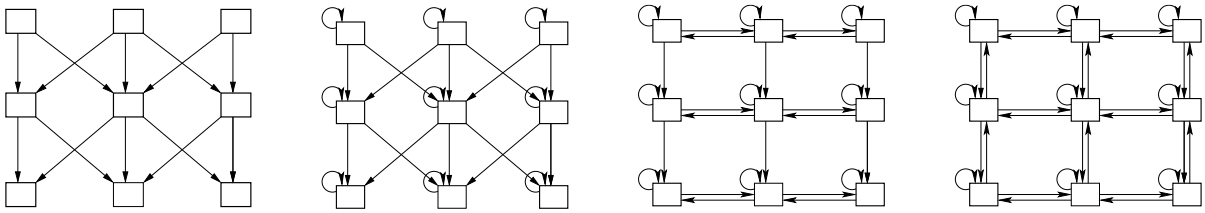
- *Contention Optimizations*: maximize instruction-level parallelism. The scheduler attempts to schedule independent instructions on different ALUs.

- *Ordering optimizations*: expose critical paths in the program. The scheduler gives priority to all instructions on the critical path, and it updates critical path information after each step.

The augmented algorithm is as follows:

```
Frames = ceil (|H|/num_alus);
S = top_down_criticality_sort(hyperblock H);
foreach instruction i in sorted list S {
  foreach rs in R {
    IssueTime(rs) = ReadyTime(rs)+Contention(rs)
    CompleteTime(i,rs) = IssueTime(rs)+Latency(i)
    Score(rs) = IssueTime(rs)+Lookahead(i)*weight
  }
  E = sort_reservation_stations(R);
  Slot(i) = prioritize(E);
  S = top_down_criticality_sort(H-{i});
}
```

We first set the number of scheduling frames at the minimum required number. For example, a block of 150 instructions would require 3 frames on an 8×8 array of ALUs. We then obtain the initial list of instructions based on the critical path metric. According to this metric, the instructions are sorted by the maximum depth of any descendant in the dataflow graph. For example, if there are only two dataflow chains in a block, $A \rightarrow B \rightarrow C \rightarrow D$, and $E \rightarrow F \rightarrow G$, the criticality metric would sort these instructions



(a) GPA−M (b) GPA−MZ (c) GPA−Tbar (d) GPA−Mesh

Figure 2: Topologies exposed to the GPA scheduler.

as A, B, C, D, E, F, G, whereas classical greedy sorters would choose the order A, B, E, C, F, D, G. The advantage of criticality-based sorting is that every instruction on the critical path will be scheduled first, minimizing communication latencies along the critical path.

Next, we compute the score for every reservation station *rs*. We incorporate all locality- and contention-aware optimizations in this score. Loads and consumers of loads are placed close to the data caches by augmenting the dataflow graph with a pseudo memory instruction and fixing the placement of this instruction. For example, a dependence edge in the DFG, $A \rightarrow B$, where A is a load instruction, is changed to $A \rightarrow M \rightarrow B$, where M has a fixed schedule at a position one hop away from the rightmost column of the grid.

Since the hardware can issue at most one instruction at every ALU in a given cycle, instructions expected to be ready at the same time are placed on different nodes. The augmented scheduler performs this function by keeping track of estimated busy times of each ALU. $ReadyTime(rs)$ is the same as the score computed in the basic algorithm described in Section 4.1. The term $Contention(rs)$ denotes any expected additional delay cycles at the ALU due to contention.

The greedy algorithm as described thus far computes scores based only on past history, namely the schedule of prior placed instructions. However, such an algorithm may perform poorly if instructions that produce register outputs are scheduled far away from the register file, because a block cannot be committed until all register outputs have been committed. We avoid this problem by incorporating a lookahead factor into the score.

$$Lookahead(i) = \frac{distance\ to\ child}{candidate\ row} + \frac{candidate\ row}{distance\ to\ child}$$

For dataflow chains that lead to a register output, this metric attempts to simultaneously choose slots further away from the registers for instructions early in the dataflow chain and to choose slots closer to the registers for instructions later in the dataflow chain.

The algorithm then selects the reservation station with the lowest score. The remaining unscheduled instructions are re-sorted if any critical paths have changed, and the whole procedure repeats until all instructions have been scheduled.

5. EVALUATION OF TOPOLOGICAL OPTIMIZATIONS

In an SPDI model, the scheduler places instructions at ALUs (nodes) based upon the node's availability and the location of the resources with which it must communicate. An interconnection network with rich topology, such as a crossbar, enables the schedule to minimize the latency due to hops in the network. However, a fully connected network requires each router to have a multitude of ports, which reduces router speed. While other restrictions on instruction placement may reduce the number of bits required to encode an instruction, this section examines the effect of routing topology on scheduler's ability to exploit concurrency.

5.1 Topology and Scheduling Trade-offs

Figure 2 shows the four topologies that we consider. We adapted the scheduling algorithm previously described for each topology described below. In every case, the scheduler begins placing instructions in the first frame of the upper right-hand node in the ALU grid, since it is close to both the caches (on the right) and the register files (on the top of the grid).

GPA-M: The GPA-M interconnect, shown in Figure 2a bears the most resemblance to a VLIW architecture, in which VLIW instructions are essentially mapped to rows, with the following instruction mapped to the succeeding row. In each row, all instructions belonging to the same frame must be independent and dependent instructions are placed in lower rows. To simplify the router, each node is connected to only the three nodes directly below it in the row (below, left, and right). When a VLIW instruction is mapped to the bottom row, the following VLIW instruction is mapped to the top row in the next frame. *Express channels* made of fat wires route results from the last row to the top row at high speed. The schedule for GPA-M is effectively an unrolled VLIW schedule, in which the VLIW words are converted from the time dimension to being unrolled along the rows and frames.

The scheduler takes the greedily-sorted list and places the independent instructions in the rightmost slot of the earliest packet possible. The scheduler searches each row in succession to see if all dependences have been satisfied and if the node is reachable from its parents. If no nodes are available in any of the rows, the scheduler tries to place the instruction in the ensuing frame. To minimize latency, the scheduler seeks to place dependent instructions in adjacent rows and columns. As many frames are allocated as necessary to schedule the entire hyperblock.

GPA-MZ: The GPA-MZ interconnect, shown in Figure 2b, enhances GPA-M by permitting an producing instruction to forward its result to a consumer mapped to the same ALU. This strategy allows routing delays between adjacent rows to be eliminated for many producer-consumer pairs, at the cost of an additional bypass path in the hardware. In the scheduling algorithm the class of legal reservation stations are extended to include the node on which an instruction's parent resides, as long as that node is reachable from all of the instruction's other parents. The earliest reservation station remains the highest logical row in the schedule. For example, if an instruction i had parents placed in row 3 and row 4, then the best position would be a different frame on the same node as the parent in row 4, assuming it was reachable from the parent in row 3.

To determine the number of frames required for the schedule, the scheduler sets the initial frame count to be the minimum required to hold all of the instructions in the hyperblock. For exam-

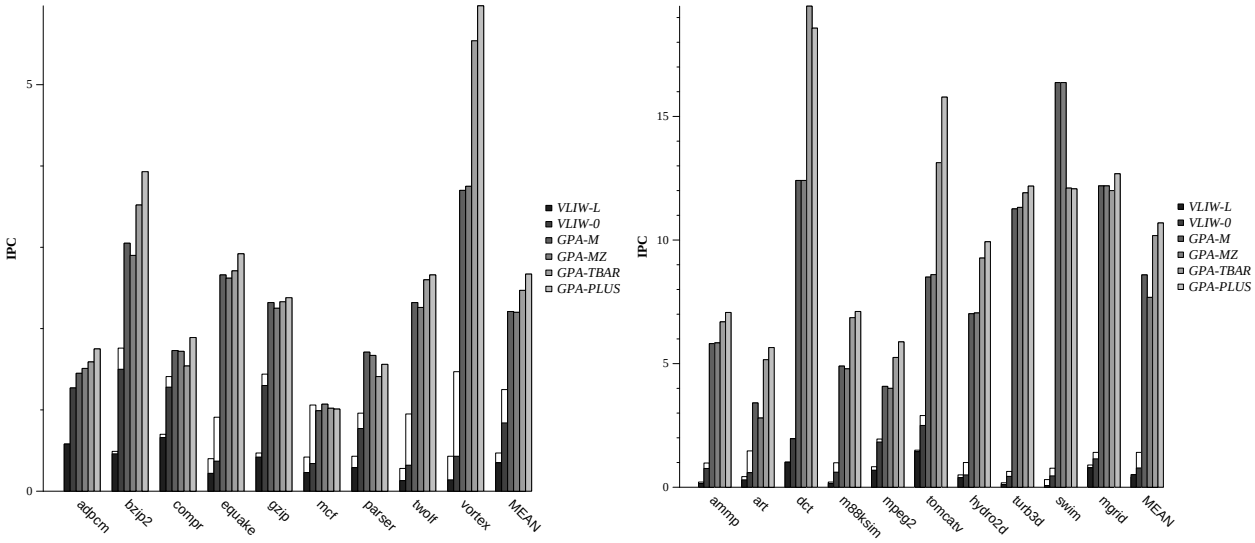


Figure 3: Performance of a VLIW and a GPA with different routers and schedulers.

ple, scheduling a 133 instruction hyperblock would start with three frames, which can be used for Z-dimension instruction placement. If the scheduler is unable to find a legal node to place the instruction (because of sparse allocation in the highest rows), the number of allocated frames to the block is incremented, as shown in the previous algorithm, and the scheduler starts from scratch with the original list.

GPA-Tbar: The GPA-Tbar topology, shown in Figure 2c, has the same number of router ports as GPA-MZ (including the Z-dimension bypass), but instead connects to the nodes right, left, and down from the producing node. This topology permits dependence chains to be routed horizontally and along the frames before being routed downward, potentially increasing the utilization of the array for blocks that have long, narrow DFGs. Since dependent instructions can be placed within the same row, the highest row is not necessarily the best choice. For example, placing a consumer one hop down from its parent in the next row results in fewer hops than placing it three hops to the left of its parent. The best reservation station is obtained using the algorithm described in 4.

GPA-Mesh: The GPA-Mesh topology, shown in Figure 2d, augments GPA-Tbar with "upward" router link to each node, allowing each node to send an operand to any of its nearest Manhattan neighbors. This topology is the most flexible of those we evaluate, allowing a consumer to be placed anywhere, regardless of the location of its parents. The advantages of this organization are that hyperblocks can be packed into the minimal number of frames because the scheduler never fails to find a legal assignment. The routers in GPA-Mesh require an additional port and are likely to run slightly slower than GPA-Tbar. The criteria to select a node from the legal candidate nodes is identical to that in the GPA-Tbar configuration.

5.2 Evaluation Methodology

The execution substrate is an 8x8 array of ALUs, which can also be treated as a 64-wide VLIW processor. We use the Trimaran compiler tool set to parse the application source code and apply aggressive VLIW optimizations. The Trimaran compiler is based on the Illinois Impact compiler [4]. It produces code in the intermediate Elcor format, based on the Hewlett-Packard PD [30] instruction set. In addition to the usual set of classic optimizations, Trimaran incorporates many VLIW-specific optimizations such as control flow profiling, trace scheduling, loop unrolling and loop peeling, software pipelining with modulo scheduling, speculative hoisted loads, and predication with both acyclic scheduling of hyperblocks and control height reduction.

We wrote a scheduler that converts the Elcor output into VLIW and GPA code, scheduled for the underlying target architectural model. For VLIW code, the scheduling pass uses top-down greedy ordering, placing the most critical instructions first in each 64-instruction instruction word, eventually either filling the long word or running out of parallel instructions. In experiments where we modeled no inter-ALU communication latency, any instruction can be assigned to any ALU with no penalty. With 0.5 cycle intra-ALU communication latencies, instructions must be placed to provide enough time for the instruction's operands to reach it before the instruction word executes. If all nodes sufficiently close to the parent have already been assigned instructions, the instruction is placed in a later long instruction word. If an instruction can be mapped to multiple candidate ALUs, the ALU closest to the parent is chosen. Generating schedules for the hybrid GPA uses the algorithm described in Section 4.

To compare performance across different execution models, substrates, and schedules, we measured instructions per cycle (IPC) using a custom detailed, cycle-accurate simulator. This tool models both VLIW and GPA microarchitectures, including realistic latencies through the processor, and simulates instruction fetching, branch prediction, the cache hierarchy, contention for ALUs, register file accesses, and branch mispredictions. We assumed 64KB primary caches, and a 2MB level-two cache.

In this study, we use a subset of the SPEC2000 [28] and the Mediabench [18] benchmark suite. The Trimaran front end currently compiles only C benchmarks, so we converted a number of the SPECFP benchmarks to C, and present results for all of the SPEC benchmarks that the Trimaran tools compiled successfully.

5.3 Scheduler vs. Topology Results

These experiments use the basic scheduler without the placement heuristics, which are not applicable to a VLIW model. We add these to the scheduler in the next section. Figure 3 shows the performance of these four instruction placement schemes, broken down into integer benchmarks on the left-hand graph and floating-point

	Base	C	CR	CRA	CRAL	CRALO	RALO
adpcm	1.76	1.70	1.71	1.66	1.84	1.80	1.77
ammp	7.07	7.08	7.24	7.09	7.33	7.32	7.19
art	5.63	5.44	5.64	5.75	5.68	5.96	6.03
bzip2	3.95	3.97	4.01	4.00	4.22	4.24	4.26
compr	1.89	1.92	1.92	1.95	2.01	1.96	2.02
dct	18.57	19.53	19.93	21.11	21.18	20.87	19.95
equake	2.93	3.02	2.95	2.95	2.94	3.18	2.97
gzip	2.38	2.41	2.43	2.43	2.55	2.57	2.68
hydro2d	9.68	8.47	9.22	9.56	9.62	9.47	9.75
m88ksim	7.11	6.96	7.58	7.76	8.30	7.76	7.57
mcf	1.01	1.12	1.12	1.10	1.20	1.21	1.05
mgrid	12.68	11.87	11.91	11.82	17.91	17.94	15.05
mpeg2	5.88	5.91	5.82	6.02	6.26	6.46	6.23
parser	1.56	1.56	1.58	1.58	1.66	1.66	1.69
swim	12.26	21.21	20.24	22.22	19.69	18.74	18.70
tomcatv	15.92	14.38	17.83	18.16	16.83	18.26	18.40
turb3d	12.20	13.07	14.32	15.18	14.89	14.65	14.56
twolf	2.66	2.55	2.56	2.54	2.47	2.70	2.76
vortex	5.97	6.11	5.74	5.73	6.70	6.63	7.06
MEAN	6.90	7.28	7.58	7.81	8.05	8.08	7.88

Table 1: Performance improvements from scheduler optimizations.

benchmarks on the right. The leftmost two bars for each benchmark present the results for a 64-wide VLIW machine. VLIW-L shows the IPC of a realistic VLIW in which the 0.5 cycle per hop routing delays are exposed to the scheduler. VLIW-0 shows the IPC of an ideal VLIW in which all inter-ALU communication is free, essentially exposing all the parallelism that the compiler is able to find in the DFG. The dark lower parts of the bars show the IPC when the cache hierarchy is modeled, while the white caps show the IPC if all memory references hit in the L1 caches. Note, however, that the VLIW models are all somewhat optimistic as we assume an infinite register set requiring no spills or restores, and instantaneous lock-step synchronization across all ALUs. Implementing such lock-step synchronization is not scalable due to the wiring delays across the distributed array of ALUs.

Both inter-ALU latencies and cache misses each cause approximately a 50% loss in exposed VLIW parallelism. The ideal VLIW (no communication latency and no cache misses) achieves nearly half of the ILP of the GPA-M topology. However, when cache misses and scheduling latencies are included, the VLIW machine achieves only one eighth the performance of a Grid Processor with comparable width on the integer benchmarks, and significantly less on the floating-point benchmarks.

GPA-MZ actually performs slightly worse than the GPA-M. Despite the reduced latency when placing dependent operations on the same node in different frames, the Z-routing capability causes the scheduler to concentrate the instructions in the upper-right-hand nodes. The restrictive GPA-MZ topology prevents dependent instructions from being spread to the left hand columns, leading to poor load balancing.

GPA-Tbar provides a large boost over GPA-MZ, because the horizontal routing channels allows the scheduler to more easily spread the instructions across the columns. With this routing topology, the scheduler can minimize the communication along the critical path, and place less critical dependence chains on longer paths toward the left of the grid. Furthermore, the scheduler can place a block with a single long chain of dependent instructions across the entire grid, minimizing the communication latency by first employing all of the frame slots of a single node before moving horizontally or vertically to the next adjacent node along a Hamiltonian path. The T-bar causes a large performance drop in the *swim* benchmark, since the scheduler places too many independent instructions on the same node.

Finally, the results show additional performance gains from the GPA-Mesh configuration. While upwards routing does not decrease communication latencies, it does permit a schedule to fit completely in the minimal number of frames. This capability permits the densest possible mapping of instructions to frames, allowing more speculative hyperblocks to be mapped onto the grid. These results indicate that the advantages of a richer network topology that provides the scheduler with more flexibility likely outweighs the disadvantages of more router ports. The GPA-Mesh shows an average of 20% performance improvement over the less flexible, VLIW-like GPA-M organization.

6. SCHEDULER OPTIMIZATIONS

In this section, we evaluate the scheduler optimizations described in Section 4 for the GPA-Mesh configuration. Recall that those optimizations try to better match the static schedules to the actual latencies experienced by the critical path at run-time.

6.1 Evaluation of Schedule Optimizations

Table 1 shows the performance results of the optimizations in different combinations. The first column, labeled *Base*, refers to our baseline GPA scheduler that is similar to a greedy VLIW scheduler. Successive columns correspond to a scheduler with one or more of the following additions to baseline scheduler: C = order by critical path, R = Recompute critical paths after each placement, A = model contention at each ALU, L = schedule loads and consumers of loads closer to data caches, and O = use lookahead to schedule instructions that produce register outputs closer to the register file.

The first set of experiments examine the effect of instruction priority in the scheduling algorithm by comparing the greedy ordering, (Base), the criticality ordering (C), and criticality with recomputation of the critical path during scheduling (CR). In the absence of other optimizations, instruction priority is not a significant factor as greedy out-performs the critical-path order in three benchmarks, while the critical-path order provides significantly higher performance on another three, improving performance by nearly 80% in *swim*. However, recomputing the critical paths improves criticality ordering, yielding the best performance on 12 of the 19 benchmarks and averaging 9% improvement over baseline and 4% improvements over the critical-path ordering.

Augmenting the scheduler with a contention model to improve load balance across the ALUs (CRA) tends to improve performance.

bench	adpcm	ammp	art	bzip2	compress	dct	equake	gzip	hydro2d	m88ksim
Dense IPC	1.80	7.32	5.96	4.24	1.96	20.87	3.18	2.57	9.47	7.76
Avg. frames	1.7	3.0	1.6	1.7	1.1	3.3	1.5	1.9	3.9	1.5
Sparse IPC	1.82	6.71	6.00	4.34	2.11	18.57	2.90	2.70	10.55	7.26
Avg. frames	3.4	6.0	3.2	3.3	2.1	6.6	3.0	3.7	7.9	4.0
Infinite IPC	1.77	6.32	5.26	4.07	2.07	17.44	2.84	2.48	9.84	5.69
Avg. frames	6.6	11.2	7.7	6.6	4.7	9.4	4.5	8.4	10.3	5.5
bench	mcf	mgrid	mpeg2	parser	swim	tomcatv	turb3d	twolf	vortex	mean
Dense IPC	1.21	17.94	6.46	1.66	18.74	18.26	14.65	2.70	6.63	8.07
Avg. frames	1.39	3.95	2.26	1.10	6.19	3.79	3.49	1.64	1.36	2.44
Sparse IPC	1.22	13.45	5.71	1.73	16.99	15.72	12.59	2.81	6.41	7.35
Avg. frames	2.7	7.9	4.5	2.1	11.8	7.6	7.0	3.3	2.7	4.82
Infinite IPC	1.25	6.22	4.61	1.60	19.17	14.83	11.03	2.49	4.57	6.50
Avg. frames	7.6	18.8	12.5	4.7	9.0	8.3	9.5	8.4	6.0	8.39

Table 2: Trade-offs of scheduling for utilization versus communication

Recall that this optimization explicitly attempts to migrate independent instructions to different ALUs. While this optimization is not important for integer programs which exhibit little parallelism, it affects performance significantly on some floating point benchmarks. Averaged across the entire benchmark suite, this optimization improves performance by 2.5% over critical path recomputation (CR).

Finally we apply the locality-aware optimizations that consider the placement of load instructions (CRAL) and instructions that write the register file (CRALO). As seen in the Table 1, optimizing for loads and consumers of loads consistently provides better performance with large gains on 6 benchmarks, indicating the importance of such optimizations. Optimizing the placement of instructions that write the register file, on the other hand, does not have much affect on performance in the presence of the load placement optimization.

6.2 Code Density Optimizations

In this section, we evaluate the effect of using different number of frames to schedule a block. Using the minimum number of frames allows the scheduler to densely pack the instructions. With the hardware providing only a finite number of reservation stations at each ALU, a dense packing enables several speculative blocks to be mapped and executed, allowing a large window of instructions to extract instruction level parallelism. Dense schedules also have the benefit of good instruction memory performance. However, providing more frames to a block creates more opportunities to schedule critical path instructions on the same ALU, thus minimizing communication latencies along the critical path.

Table 2 explores this trade-off. The first row shows the IPC when the scheduler is allowed to use only the minimum number of frames for each block it schedules. The second line in the same row shows the average number of frames used by the blocks that occur during execution. For example, the benchmark *equake* exhibits an IPC of 3.18, when using dense schedules which utilize 1.5 frames per block on the average. The second row shows the corresponding results, when the scheduler uses twice the minimum number of frames, and the last row corresponds to the case, when the scheduler is free to use as many frames as the length of the longest dataflow chain.

As can be seen, using dense schedules provides superior performance on most benchmarks. On a small set of benchmarks (*bzip2, compress*), we notice that performance improves marginally with sparse schedules. On such benchmarks, we observed low branch prediction rates, and hence the available reservation stations were not fully utilized during execution. The sparse schedules benefitted from using those unused reservation stations, minimizing the communication along the critical path. When we use still sparser schedules (Infinite), the performance further deteriorates, a fact exacerbated in some floating point benchmarks that are inherently latency-tolerant.

7. SCHEDULING FOR COMPATIBILITY

A major drawback of traditional VLIW architectures, and SPSI machines in general, is a lack of object code compatibility across generations. If issue width of the hardware or the runtimes latencies used by the compiler change, the binaries must be rescheduled. Since static issue architectures do not dynamically check instruction dependencies or change instruction placements, the compiler must make static assumptions about the topology and latencies to ensure correct execution.

Conversely, SPDI architectures do not enforce a rigorous static schedule and still produce correct program execution even if the dynamic latencies differ from those used at compile time. In this section, we examine the sensitivity of the performance of an SPDI machine to exact knowledge of dynamic communication latencies. Our results show that while the best schedules are usually produced when the static and dynamic latencies match, the performance degradation when they mismatch is typically less than 10%. We also describe a method of dynamically mapping a schedule created for a large SPDI machine onto a machine with fewer ALUs. We evaluate performance degradations resulting from a mismatch in issue width and show that a single schedule may run effectively across machines with different issue widths, requiring no translation or recompilation.

7.1 Sensitivity to Wire Latencies

We evaluated the sensitivity of the scheduler to specific wire delays by scheduling for a fixed delay and then simulating that schedule on an implementation with different delays. We then compared those to our original results in which scheduler knows the communication delays precisely. Table 3 shows the performance, measured in IPC, of programs scheduled for an 8x8 GPA-Mesh configuration, with two rows for each of four single-hop communication latencies ranging from 0.5–3 cycles. The top row in each pair is the absolute IPC when each benchmark is scheduled and executed with matching static and dynamic latencies. The second row shows the difference in IPC when the program is scheduled for a one-cycle latency but run using the varying latencies. Of course, only one row is shown for the one cycle hop latency since the static and dynamic latencies always match.

Negative numbers in ΔIPC indicates that performance degrades when the static and dynamic latencies mismatch, while positive

scheduled latencies	adpcm	ammp	art	bzip2	compress	dct	equake	gzip	hydro2d	m88ksim
IPC - 0.5 cycles	1.80	7.32	5.95	4.24	1.96	20.87	3.09	2.57	9.47	7.76
ΔIPC (%)	0	0	0.2	0	0	0	2.9	0	0	0
IPC - 1.0 cycle	1.38	5.69	4.92	3.27	1.52	15.35	2.70	1.93	6.91	5.74
IPC - 2.0 cycles	0.93	3.81	3.42	1.95	1.05	10.73	1.94	1.27	4.41	3.56
ΔIPC (%)	-2.2	-1.8	4.1	9.7	-1.9	-2.1	2.6	-1.6	-3.6	0.6
IPC - 3.0 cycles	0.66	3.02	2.66	1.52	0.75	6.93	1.55	0.84	3.22	2.50
ΔIPC (%)	-4.5	-10.6	0.4	-0.7	-4	3.3	1.3	2.4	-6.5	0.4
	mcf	mgrid	mpeg2	parser	swim	tomcatv	turb3d	twolf	vortex	mean
IPC - 0.5 cycles	1.22	17.94	6.46	1.66	18.74	18.26	14.65	2.70	6.58	8.06
ΔIPC (%)	-0.8	0	0	0	X	X	0	0	0.8	0.2
IPC - 1.0 cycle	0.94	13.22	4.75	1.3	10.35	12.63	9.20	2.19	5.21	5.7
IPC - 2.0 cycles	0.59	4.58	2.83	1.01	5.65	6.28	5.14	1.53	3.95	3.17
ΔIPC (%)	8.5	69.9	5.7	-12.9	3.9	2.2	0	1.9	-2.7	4.2
IPC - 3.0 cycles	0.41	4.94	1.98	0.67	4.54	3.97	3.44	1.14	2.54	2.48
ΔIPC (%)	9.8	6.3	3.0	-7.5	-11.2	9.0	2.6	0	7	0

Table 3: Sensitivity of the SPDI schedule to wire dynamic routing latencies.

numbers indicate performance improvements. For example, the IPC of *dct* sheduled and run with 2-cycle hop latencies is 10.73. When it is scheduled for 1-cycle hop latencies and run with 2-cycle latencies, the IPC drops to 10.5, a loss of 2.1%. The overall results show that if the wires are faster than those for which they were scheduled (0.5 cycles), the performance is virtually the same. If the wires are slower (2 or 3 cycles per hop), performance drops by only 2–3%. Some benchmarks, such as *bzip2* and *mgrid* are extremely sensitive to the schedule and their performance actually improves when the wires are slower than those assumed by the scheduler. This somewhat surprising result demonstrates the sensitivity of runtime performance to compile time assumptions of the hardware, and we are continuing to investigate its causes.

7.2 Sensitivity to Issue Width

SPDI architectures can achieve cross-generation compatibility by further virtualizing the nodes in the ALU array. Since the GPA-Mesh topology is completely connected, instructions may be assigned anywhere and the schedule will still produce the correct result. Virtualization is achieved by scheduling the code for a large array (for example, a 64-wide 8x8 GPA used in this section), which can then be run on a smaller array by dynamically mapping the instructions from multiple nodes in the larger virtual array to a single node in the smaller physical array. This virtualization strategy requires that enough frames are available on the physical array to store all of the instructions from the schedule of the virtual array. For example, an instruction block consuming 2 frames in an 8x8 array may consist of up to 128 instructions. Mapping this block onto a 4x4 array would require 8 frames of storage on the smaller array. The mapping function can be performed entirely in hardware by interpreting the instruction placement addresses, specified as coordinates in the X, Y, and Z dimensions of the virtual grid, differently on different size grids. As an example, an address of <1,7,1> representing row 1, column 7, frame 1 on the 8x8 virtual grid can be mapped to <0,3,7> on the 4x4 array by translating the binary addresses from <*00***1**,*11***1**,*1*> to <*00*,*11*,***11***1>.

We evaluated the performance losses incurred by running programs scheduled for a larger 8x8 array on smaller arrays. Table 4 shows the performance degradation as compared to codes explicitly scheduled for the smaller arrays. The rows labeled "IPC" show the raw instructions per clock when the scheduler knows the exact topology of the grid. The rows labeled ΔIPC show the change in performance when running the 8x8 schedule on each smaller grid size indicated. Results show that performance drops an average of 5% when running an 8x8 schedule on an 8x4 array (8 rows, 4 columns), 17% when running on a 4x4 array, and 22% when running on a 4x2 array. However, those performance drops are negligible when compared to the performance *gains* that can be achieved on programs with substantial parallelism by migrating to larger grid dimensions. Thus a compatibility path can be provided by scheduling all codes for large arrays, initially running them on small arrays, and achieving performance improvements by incrementally migrating to larger arrays until reaching the grid size for which the applications were originally scheduled.

8. CONCLUSIONS

Conventional architectures sit at opposite ends of the spectrum with regard to their demands on the scheduler. While superscalar architectures can improve some schedules through dynamic scheduling hardware and can see some benefit from good instruction schedulers, performance is ultimately constrained by the limited instruction window size. At the other end of the spectrum, VLIW architectures demand that the compiler place every instruction and schedule every latency. Such demands are unrealistic in the face of uncertain memory latencies and statically uncertain aliases. A hybrid approach that allows the scheduler to place instructions for good locality while also allowing the hardware to dynamically execute the instructions (overlapping instruction latencies and other unknown latencies) can produce better performance. Such approaches will become even more important as technology trends make communication more critical due to increased wire delays.

We have implemented and evaluated a scheduler for such an emerging architecture. Because the hardware dynamically executes the instructions, the scheduler is freed from the burden of precise scheduling constraints. Instead its job is to expose the concurrency in the instruction stream and place the instructions to minimize communication overheads. Our scheduling algorithm is able to achieve a tightly packed schedule using a minimum number instruction slots, while still minimizing latency and balancing the load across the ALUs, thus eliminating hot spots where too many independent instructions have been placed.

We have evaluated our scheduler on a 64-issue processor and examined the interplay between the hardware constraints and the scheduler's capabilities. We show that the freedom provided by a mesh interconnect topology allows the scheduler to expose more concurrency while minimizing the number of hops, resulting in a **??**% performance improvement over more restrictive topologies. We demonstrate that accounting for distances, not just between ALUs, but also to the register file and cache banks, is critical for performance. An algorithm to estimate instruction execution times

grid dimensions	adpcm	ammp	art	bzip2	compress	dct	equake	gzip	hydro2d	m88ksim
IPC - 8x8	1.80	7.32	5.96	4.24	1.96	20.87	3.18	2.57	9.47	7.76
IPC - 8x4	1.81	6.67	5.77	4.26	1.91	17.81	3.04	2.68	9.46	7.66
IPC - 4x4	1.89	5.88	5.32	4.26	2.11	12.31	2.64	2.62	7.30	6.89
IPC - 4x2	1.71	3.89	4.18	3.48	1.98	6.60	2.00	2.15	4.01	4.80
ΔIPC - 8x4 (%)	1.6	-2.4	-6.1	-0.9	7.3	-2.2	-8.55	-0.37	-5.3	-5.1
ΔIPC - 4x4 (%)	-1.6	-11.2	-5.6	-8.7	-1.4	-18.9	-10.2	-6.5	-11.2	-18.4
ΔIPC - 4x2 (%)	-7.6	-10.8	-8.9	-17.5	-13.1	-8.0	-18.5	-17.7	-14.9	-27.7

	mcf	mgrid	mpeg2	parser	swim	tomcatv	turb3d	twolf	vortex	mean
IPC - 8x8	1.21	17.94	6.46	1.66	18.74	18.26	14.65	2.70	6.63	8.07
IPC - 8x4	1.21	15.19	5.51	1.68	13.74	14.12	12.73	2.76	6.07	7.05
IPC - 4x4	1.25	8.94	5.18	1.78	9.62	9.11	9.37	2.69	5.57	5.51
IPC - 4x2	1.22	4.80	4.17	1.69	4.06	4.80	5.28	2.43	4.05	3.54
ΔIPC - 8x4 (%)	5.7	-14.9	3.0	1.8	-7.4	-0.9	-4.7	-0.7	0.8	-2.1
ΔIPC - 4x4 (%)	0.8	-26.8	-6.9	-5.6	-17.7	-10.4	-7.9	-12.0	-10.4	-10.0
ΔIPC - 4x2 (%)	-9.8	-27.3	-16.3	-14.2	-8.4	-7.7	-12.3	-17.3	-24.2	-14.9

Table 4: Sensitivity to issue width and topology

was necessary to improve load balancing and to help place independent instructions on different nodes. Finally, we show that instruction criticality is important and that iteratively updating the estimated critical path during the instruction placement process provides a 10% boost in performance over a single priority listing. Combining the strengths of static scheduling with the advantages of dynamic issue will be critical to achieve performance in emerging wire-dominated technologies.

9. REFERENCES

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster dynamically-scheduled, superscalar processors. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 337–347, December 2000.

[3] R. Canal, J. M. Parcerisa, and A. Gonzalez. Dynamic cluster assignment mechanisms. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, pages 132–142, 2000 January.

[4] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. mei W. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 266–275, May 1991.

[5] P. Craig, R. Crowell, M. Liu, B. Noyce, and J. Pieper. The Gem loop transformer. *Digital Technical Journal*, 1999.

[6] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30rd International Symposium on Microarchitecture*, pages 149–159, December 1997.

[7] J. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the Tenth Annual International Symposium on Computer Architecture*, pages 140–150, June 1983.

[8] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[9] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984.

[10] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 11–16, Palo Alto, CA, June 1986.

[11] W. Havanki, S. Banerjia, and T. Conte. Treegion scheduling for wide-issue processors. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 266–276, 1998.

[12] M. Horowitz, C.-K. K. Yang, and S. Sidiropoulos. High-speed electrical signaling: overview and limitations. In *IEEE Micro*, pages 12–24, January 1998.

[13] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the ia-64 architecture. *IEEE Micro*, 20(5):12–23, September/October 2000.

[14] D. R. Kerns and S. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 278–289, Albuquerque, NM, June 1993.

[15] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March 1999.

[16] G. Krishnamurthy, E. D. Granston, and E. J. Stotzer. Affinity-based cluster assignment for unrolled loops. In *Proceedings of the 2002 ACM International Conference on Supercomputing*, pages 107–116, 2002.

[17] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, GA, June 1988.

[18] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[19] G. Lindenmaier, K. S. McKinley, and O. Temam. Load scheduling with profile information. In A. Bode, T. Ludwig, and R. Wismüller, editors, *Euro-Par 2000 – Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 223–233, Munich, Germany, Aug. 2000. Springer-Verlag.

[20] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, June 1992.

[21] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 40–51, December 2001.

[22] K. Palem and B. Simons. Scheduling time-critical instructions on risc machines. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–280, 1990.

[23] Y. Qian, S. Carr, and P. Sweany. Optimizing loop

performance for clustered vliw architectures. In *The 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 271–280, Charlottesville, VA, Sept. 2002.

[24] N. Ranganathan and M. Franklin. An empirical study of decentralized ILP execution models. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–281, October 1998.

[25] B. Rau. Dynamically scheduled VLIW processors. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 80–90, December 1993.

[26] J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: optimal vs. heuristic methods in a production compiler. In *Proceedings of the ACM SIGPLAN '96 conference on Programming language design and implementation*, pages 1–11, 1996.

[27] The national technology roadmap for semiconductors. Semiconductor Industry Association, 1999.

[28] Standard Performance Evaluation Corporation. *SPEC CPU 2000, http://www.spec.org/osg/cpu2000*, April 2000.

[29] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic unit s. *IBM Journal*, 11:25–33, January 1967.

[30] V.Kathail, M.Schlansker, and B.R.Rau. Hpl-pd architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard Laboratories, February 2000.