Convergent Scheduling

Walter Lee, Diego Puppin, Shane Swenson, Saman Amarasinghe
Massachusetts Institute of Technology
{walt, diego, sswenson, saman}@cag.lcs.mit.edu

Abstract

Convergent scheduling is a general framework for cluster assignment and instruction scheduling on spatial architectures. A convergent scheduler is composed of independent passes, each implementing a heuristic that addresses a particular problem or constraint. The passes share a simple, common interface that provides spatial and temporal preference for each instruction. Preferences are not absolute; instead, the interface allows a pass to express the confidence of its preferences, as well as preferences for multiple space and time slots. A pass operates by modifying these preferences. By applying a series of passes that address all the relevant constraints, the convergent scheduler can produce a schedule that satisfies all the important constraints. Because all passes are independent and need to understand only one interface to interact with each other, convergent scheduling simplifies the problem of handling multiple constraints and co-developing different heuristics. We have applied convergent scheduling to two spatial architectures: the Raw processor and a clustered VLIW machine. It is able to successfully handle traditional constraints such as parallelism, load balancing, and communication minimization, as well as constraints due to preplaced instructions, which are instructions with predetermined cluster assignment. Convergent scheduling is able to obtain an average performance improvement of 21% over the existing space-time scheduler of the Raw processor, and an improvement of 14% over state-of-the-art assignment and scheduling techniques on a clustered VLIW architecture.

1 Introduction

Instruction scheduling on microprocessors is becoming a more and more difficult problem. In almost all practical instances, it is NP complete, and it often faces multiple contradictory constraints. For superscalars or VLIWs, the two primary issues are parallelism and register pressure. Code sequences that expose more instruction level parallelism (ILP) also have longer live ranges and higher register pressure. To generate good schedules, the instruction scheduler must somehow exploit as much ILP as possible without leading to a large number of register spills.

On spatial architectures, instruction scheduling is even more complicated. Examples of spatial architectures include clustered VLIWs, Raw [24], Grid processors [22], and ILDPs [12]. Spatial architectures are architectures that distribute their computing resources. Communication between distant resources can incur one or more cycles of delays. On these architectures, the instruction scheduler has to partition instructions across the computing resources. Thus, instruction scheduling becomes both a spatial problem and a temporal problem.

To make partitioning decisions, the scheduler has to understand the proper tradeoff between parallelism and locality. Figure 1 shows an example of this tradeoff. Spatial scheduling by itself is already a more difficult problem than temporal scheduling, because a small spatial mistake is generally more costly than a small temporal mistake. If a critical instruction is scheduled one cycle later than desired, only one cycle is lost. But if a critical instruction is scheduled one unit of distance farther away than desired, cycles can be lost from unnecessary communication delays, additional communication resource contention, and increase in register pressure.

In addition, some instructions on spatial architectures have specific spatial requirements. These requirements arise from two sources. First, some loads and stores instructions must access memory banks on specific clusters, either for correctness or for performance reasons [2, 6, 13]. Second, when a value is live across scheduling regions, its definitions and uses must be mapped to a consistent cluster [14]. We call instructions with these spatial requirements *preplaced instructions*. A good scheduler must be sensitive to

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

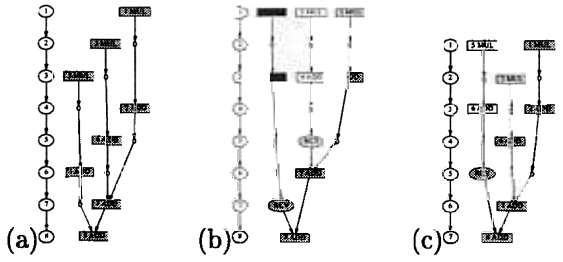

Figure 1. An example of tradeoff between parallelism and locality on spatial architectures. Each node color represents a different cluster. Consider an architecture with three clusters, each with one functional unit, where communication takes one cycle of latency due to the "receive" instruction. In (a), conservative partitioning that maximizes locality and minimizes communication leads to an eight-cycle schedule. In (b), aggressive partitioning has high communication requirements and leads to an eight-cycle schedule. The optimal schedule, in (c), takes only seven cycles: it is a careful tradeoff between locality and parallelism.

constraints imposed by preplaced instructions in order to generate a good schedule.

A scheduler also faces difficulties because different heuristics work well for different types of graphs. Figure 2 depicts representative data dependence graphs from two ends of a spectrum. In the graphs, nodes represent instructions and edges represent data dependences between instructions. Graph (a) is typical of graphs seen in non-numeric programs, while graph (b) is representative of graphs coming from applying loop unrolling to numeric programs. Consider the problem of scheduling these graphs onto a spatial architecture. Long, narrow graphs are dominated by a few critical paths. For these graphs, critical-path based heuristics are likely to work well. Fat, parallel graphs have coarse grained parallelism available and many critical paths. For these graphs it is more important to minimize communication and exploit the coarse-grain parallelism. To perform well for arbitrary graphs, a scheduler may require multiple heuristics in its arsenal.

Traditional scheduling frameworks handle conflicting constraints and heuristics in an *ad hoc* manner. One approach is to direct all efforts toward the most serious problem. For example, modern RISC superscalars can issue up to four instructions and have tens of registers. Furthermore, most integer programs tend to have little ILP. Therefore, many RISC schedulers

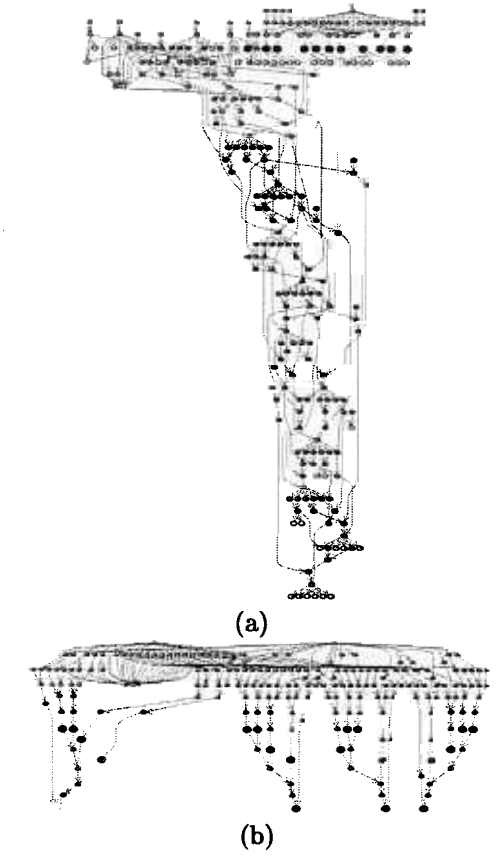

Figure 2. Different data dependence graphs have different characteristics. Some are thin and dominated by a few critical paths (a), while others are fat and parallel (b).

focus on finding ILP and ignore register pressure altogether. Another approach is to address the constraints one at a time in a sequence of phases. This approach, however, introduces phase ordering problems, as decisions made by the early phases are based on partial information and can adversely affect the quality of decisions made by subsequent phases. A third approach is to attempt to address all the problems together. For example, there have been reasonable attempts to perform instruction scheduling and register allocation at the same time [21]. However, extending such frameworks to support preplaced instructions is difficult – no such extension exists today.

This paper presents *convergent scheduling*, a radical departure from traditional scheduling methods. Convergent scheduling is a general scheduling framework that makes it easy to specify arbitrary constraints and scheduling heuristics. Figure 3 illustrates this framework. A convergent scheduler is composed of indepen-

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

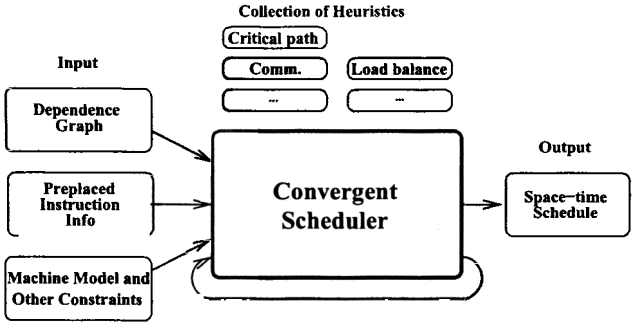

Figure 3. Convergent schedule framework.

dent phases. Each phase implements a heuristic that addresses a particular problem such as ILP or register pressure. Multiple heuristics may address the same problem.

All phases in the convergent scheduler share a common interface. The input or output to each phase is a collection of spatial and temporal preferences of instructions. A phase operates by analyzing the current preferences and modifying them. As the scheduler applies the phases in succession, the preference distribution converges to a final schedule that incorporates the preferences of all the constraints and heuristics. Logically, preferences are specified as a three-input function that maps an instruction, space, and time triple to a weight.

The contributions of this paper are:

- a novel interface between scheduling passes based on weighted preferences;
- a novel approach to address the combined problems of cluster assignment, scheduling, and register pressure,

  the formulation of a set of powerful heuristics to address both general constraints and architecture-specific issues,
- a demonstration of the effectiveness of convergent scheduling compared to traditional techniques.

The rest of this paper is organized as follows. Section 2 introduces convergent scheduling and uses an example to illustrates how it works. Section 3 describes the convergent scheduling interface between passes. Section 4 describes the collection of passes currently implemented in our framework. Section 5 presents experimental results for two architectures: a clustered VLIW and the Raw processor [24]. Section 6 provides related work. Section 7 concludes.

2 Convergent scheduling

In the convergent scheduling framework, passes communicate their choices as changes in the relative preferences of instructions for clusters and time slots.[1] The spatial and temporal preferences of each instruction are represented as weights in a preference map; a pass influences the scheduling of an instruction by changing these weights. When convergent scheduling completes, the slot in the map with the highest weight is designated as the *preferred slot*, which includes both a preferred cluster and a preferred time. The instruction is assigned to the preferred cluster; the preferred time is used as the instruction priority for list scheduling.

Different heuristics work to improve the schedule in different ways. The *critical path strengthening* heuristic (PATH), for example, expresses a preference to keep all the instructions in critical paths together in the same cluster. The *communication minimization* heuristic (COMM) tries to keep dependent neighboring instructions in the same cluster. The *preplacement* heuristic (PLACE) prefers that preplaced instructions and their neighbors are placed on the clusters selected by the preplaced instructions. The *load balance* heuristic (LOAD) changes reduces the preferences on highly loaded clusters cluster and increases them on the less loaded ones. Other heuristics will be introduced in Section 4.

Figure 4 shows how convergent scheduling operates on a small code sequence from fpppp. Initially, the weights are evenly distributed, as shown in (b). We apply the *noise introduction* heuristic (NOISE) to break symmetry, resulting in (c). This heuristic helps increase parallelism by distributing instructions to different clusters. Then, we run *critical path strengthening* (PATH), which increases the weight of the instructions in the critical path (*i.e.*, instructions 23, 25, 26, etc.) in the first cluster (d). Then we run the *communication minimization* (COMM) and the *load balance* (LOAD) heuristics, resulting in (e). These heuristics lead to several changes: the first few instructions are pushed out of the first cluster, and groups of instructions start to assemble in specific clusters (*e.g.*, instructions 19, 20, 21, and 22 in cluster 3).

Next, we run PLACE and PLACEPROP, which bias instructions using information from preplaced nodes. The result is shown in (f). The pass causes a lot of disturbances: preplaced instructions strongly attract neighbors of preplaced instructions to their clusters.

[1] In the paper, the following terms will be used interchangeably: *phases* and *passes*, *tile* and *cluster*, and *cycle* and *time slot*.

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

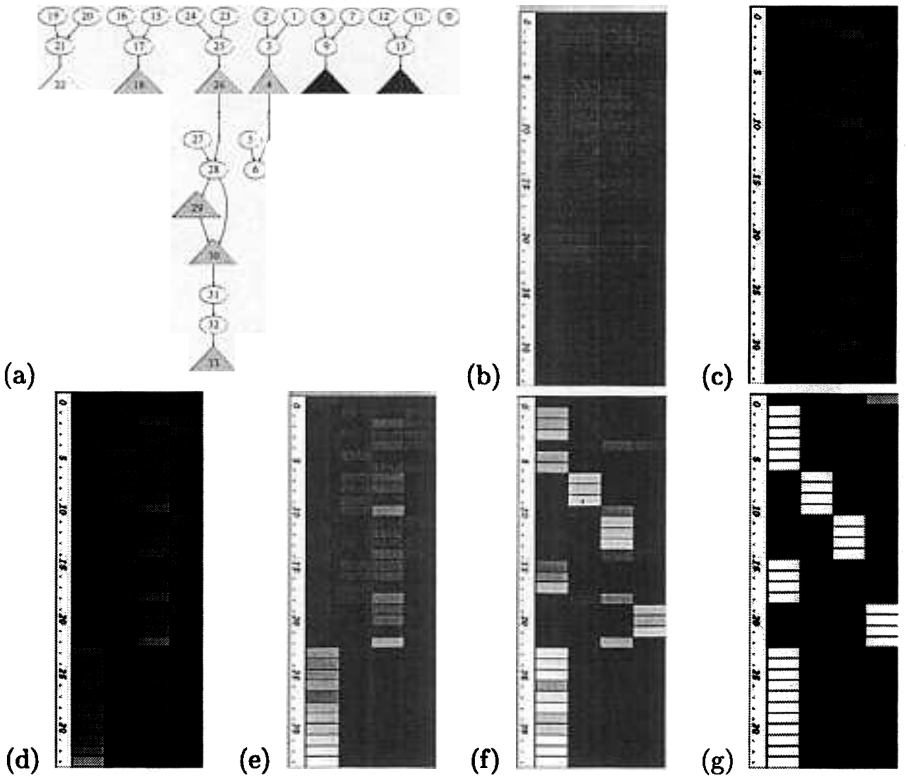

Figure 4. Convergent scheduling operates on a code sequence from fpppp. Figure (a) shows the data dependence graph representation of the scheduled code. Nodes represent instructions; edges represent dependences between instructions. Triangular nodes are preplaced, with different shades corresponding to different clusters. Figures (b)-(g) show how the convergent schedule is modified by a series of passes. This example only illustrates space scheduling, not time scheduling. Each figure in Figures 4b-g is a cluster preference map. A row represents an instruction. The row numbers correspond to the instruction numbers in (a). A column represents a cluster. The color of each entry represents the level of preference an instruction has for that cluster. The lighter the color, the stronger the preference.

Observe how the group 19–22 is attracted to cluster 4. Finally we run *communication minimization* (COMM) another time. The final schedule is shown in (g).

Convergent scheduling has the following features:

Its scheduling decisions are made *cooperatively* rather than *exclusively*.

2. The interface allows a phase to express confidence about its decisions. A phase needs not make a poor and unrecoverable decision just because it has to make a decision. On the other side, any pass can strongly affect the final choice if needed.

3. Convergent scheduling can naturally recover from a temporary wrong decision by one phase. In the example, when we apply noise to (b), most nodes are initially moved away from the first cluster. Subsequently, however, nodes with strong ties to cluster one, such as nodes 1–6, are eventually moved back, while nodes without strong ties, such as node 0, remain away.

4. Most compilers allow only very limited exchange of information among passes. In contrast, the weight-based interface to convergent scheduling is very expressive.

5. The framework allows a heuristic to be applied multiple times, either independently or as part of an iterative process. This feature is useful to provide feedback between phases and to avoid phase

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

ordering problems.

6. The simple interface (preference maps) between passes makes it easy for the compiler writer to handle new constraints or design new heuristics. Phases for different heuristics are written independently, and the expressive, common interface reduces design complexity. This offers an easy way to retarget a compiler and to address peculiarities of the underlying architecture. If, for example, an architecture is able to exploit auto-increment on memory-access with a specific instruction, one pass could try to keep together memory-accesses and increments, so that the scheduler will find them together and will be able to exploit the advanced instructions.

3 Convergent interface

This section describes the convergent scheduling interface between passes. The passes themselves will be presented in Section 4.

Convergent scheduling operates on individual scheduling units, which may be basic blocks, traces [7], superblocks [10], or hyperblocks [19], or treegions [9]. It stores preferences in a three dimensional matrix $W_{i,c,t}$, where i spans over all instructions in the scheduling unit, c spans over the clusters in the architecture, and t spans over time. We allocate as many time slots as the critical-path length (CPL).

Initially, all the weights are distributed evenly. A pass examines the dependence graph and the weight matrix to determine the characteristics of the preferred schedule so far. Then, it expresses its preferences by manipulating the preference map. Passes are not required to perform changes that affect the preferred schedule. If they are indifferent to one or more choices, they can keep the weights the same.

Let i spans over instructions, c over clusters, t over time-slots. The following invariants are maintained:

$$\forall i,t,c : 0 \leq W_{i,t,c} \leq 1$$

$$\forall i : \sum_{t,c} W_{i,t,c} = 1$$

Given an instruction i, we define the following:[2]

$$\text{preferred\_time(i)} = \arg\max \left\{ t : \sum_{c} W_{i,t,c} \right\}$$

$$\text{preferred\_cluster(i)} = \arg\max \left\{ c : \sum W_{i,t,c} \right\}$$

$$\text{runnerup\_cluster(i)} = \arg\max \left\{ \begin{array}{l} c : \sum_t W_{i,t,c}; \\ c \neq \text{preferred\_cluster(i)} \end{array} \right\}$$

$$\text{confidence(i)} = \frac{\sum_t W_{i,t,\text{preferred\_cluster(i)}}}{\sum_t W_{i,t,\text{runnerup\_cluster(i)}}}$$

Preferred values are those that maximize the sum of the preferences over time and clusters.

The confidence of an instruction measures how confident the convergent scheduler is about its current spatial assignment. It is computed as the ratio of the weights of the top two clusters.

The following basic operations are available on the weights:

- Any weight $W_{i,t,c}$ can be increased or decreased by a constant, or multiplied by a factor.
- The matrices of two or more instructions can be combined linearly to produce a matrix of another instruction. Given input instructions $i_1, i_2, ..., i_n$, an output instruction j, and weights for the input instructions $w_1, ..., w_n$ where $\sum_{k\in[1,n]} w_k = 1$, the linear combination is as follows:

  $$\text{for each } (c,t), W_{j,t,c} \leftarrow \sum_{k\in[1,n]} w_k * W_{i_k,t,c}$$

  Our current implementaion uses a simpler form of this operation, with $n = 2$ and $i_1 = j$:

  $$\text{for each } (c,t), W_{i_1,t,c} \leftarrow w_1 W_{i_1,t,c} + (1 - w_1) W_{i_2,t,c}$$

  We never perform this full operation because it is expensive. Instead, we only do it on part of the matrices, *e.g.*, only along the space dimension, or only within a small range along the time dimension.
- The system incrementally keeps track of the sums of the weights over both space and time, so that they can be determined in $O(1)$ time. It also memoizes the preferred_time and preferred_cluster of each instruction.
- The preferences can be normalized to guarantee our invariants; the normalization simply performs:

  $$\text{for each } (i,c,t), W_{i,t,c} \leftarrow \frac{W_{i,t,c}}{\sum_{t,c} W_{i,t,c}}$$

4 Collection of Heuristics

This section presents a collection of heuristics we have implemented for convergent scheduling. Each heuristic attempts to address a single constraint and only communicates with other heuristics via the weight matrix. There are no restrictions on the order or the number of times each heuristic is applied. Currently, the following parameters are selected by trial-and-error: the set of heuristics we use, the weights used

[2] The function arg max returns the value of the variable that maximizes the expression for a given set of values (while max return the value of the expression). For instance $\max\{0 \leq x \leq 2 : 10 - x\}$ is 10, and $\arg\max\{0 \leq x \leq 2 : 10 - x\}$ is 0.

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

in the heuristics, and the order in which the heuristics are run. We expect to implement more systematic heuristics selection in the future.

Whenever necessary, we run normalization at the end of every pass to ensure the invariants described in Setion 3. For brevity, this step is not included in the description below.

Initital time assignment (INITTIME) Instruction in the middle of the dependence graph cannot be scheduled before their predecessors, nor after their successors. So, if CPL is the lenght of the critical path, l_p is the length of the longest path from the top of the graph (latency of predecessor chain), and l_s is the longest path to any leaf (latency of successor chain), the instruction can be scheduled only in the time slots between l_p and $CPL - l_s$. If an instruction is part of the critical path, only one time-slot will be feasible. This pass squashes to zero all the weights outside this range.

$$\text{for each } (i, t < l_p \cup t < CPL - l_s, c),\ W_{i,t,c} \leftarrow 0$$

A pass similar to this one can address the fact that some instructions cannot be scheduled in all clusters in some architectures, simply by squashing the weights for the unfeasible clusters.

Noise introduction (NOISE) This pass introduces a small amount of noise in the weight distribution. The noise helps break symmetry and spreads instructions around to facilitate scheduling for parallelism.

$$\text{for each } (i, t, c),\ W_{i,t,c} \leftarrow W_{i,t,c} + \text{rand}()/\text{RAND\_MAX}$$

Preplacement (PLACE) This pass increases the weight for preplaced instructions to be placed in their home cluster. Since this condition is required for correctness, the weight increase is large. Given preplaced instruction i, let $cp(i)$ be its preplaced cluster. Then,

$$\text{for each } (i \in PREPLACED, t),\ W_{i,t,cp(i)} \leftarrow 100 W_{i,t,cp(i)}$$

Push to first cluster (FIRST) In our clustered VLIW infrastructure, an invariant is that all the data are available in the first cluster at the beginning of every scheduling unit. For this architecture, we want to give advantage to a schedule utilizing more the first cluster, where data are already available, versus the other clusters, were copies can be needed. We express this preference as follows:

$$\text{for each } (i, t), W_{i,t,1} \leftarrow 1.2 W_{i,t,}$$

Critical path strengthening (PATH) This pass tries to keep all the instructions on a critical path (CP) in the same cluster. If instructions in the paths have bias for a particular cluster, the path is moved to that cluster. Otherwise the least loaded cluster is selected. If different portions of the paths have strong bias toward different clusters (*e.g.*, when there are two or more preplaced instructions on the path), the critical path is broken in two or more pieces and kept locally close to the relevant home clusters. Let $cc(i)$ be the chosen cluster for the CP.

$$\text{for each } (i \in CP, t, c),\ W_{i,t,cc(i)} \leftarrow 3 W_{i,t,cc(i)}$$

Communication minimization (COMM) This pass reduces communication load by increasing the weight for an instruction to be in the same clusters where most of neighbors (successors and predecessors in the dependence graph) are. This is done by summing the weights of all the neighbors in a specific cluster, and using that to skew weights in the correct direction.

$$\text{for each } (i, t, c), W_{i,t,c} \leftarrow W_{i,t,c} \sum_{n \in \text{neighbors of } i} W_{n,t,c}$$

We have also implemented a variant of this that considers *grand-parents* and *grand-children*, and we usually run it together with COMM.

$$\text{for each } (i), W_{i,t_i,c_i} \leftarrow 2 W_{i,t_i,c_i}$$

Preplacement propagation (PLACEPROP) This pass propagates preplacement information to all instructions. For each non-preplaced instruction i, we divide its weight for each cluster c by its distance to the closest preplaced instruction in c. Let $dist(i, c)$ be this distance. Then,

$$\text{for each } (i \notin PREPLACED, t, c),\ W_{i,t,c} \leftarrow W_{i,t,c}/dist(i, c)$$

Load balance (LOAD) This pass performs load balancing across clusters. Each weight on a cluster is divided by the total load on that cluster:

$$\text{for each } (i, t, c), W_{i,t,c} \leftarrow W_{i,t,c}/load(c)$$

Level distribute (LEVEL) This pass distributes instructions at the same *level* across clusters. Given instruction i, we define $level(i)$ to be its distance from the furthest root. Level distribution has two goals. The primary goal is to distribute parallelism across clusters. The second goal is to minimize potential communication. To this end, the pass tries to distribute instructions that are far apart, while keeping together instructions that are near each other.

To perform the dual goals of instruction distribution without excessive communication, instructions on a level are partitioned into bins. Initially, the bin B_c

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

for each cluster c contains instructions whose preferred cluster is c, and whose confidence is greater than a threshold (2.0). Then, we perform the following:

LevelDistribute: int l, int g
  I_l = Instruction i | level(i) = l
  foreach Cluster c
    $I_l = I_l - B_c$
  $I_g = \{i \mid i \in I_l; \text{distance}(i, \text{find\_closest\_bin}(i)) > g\}$
  while $I_l \neq \phi$
    B = round_robin_next_bin()
    $i_{closest} = \arg\max\{i \in I_g : \text{distance}(i, B)\}$
    $B = B \cup i_{closest}$
    $I_l = I_l - i_{closest}$
    Update I_g

The parameter g controls the minimum distance granularity at which we distribute instructions across bins. The distance between an instruction i and a bin B is the minimum distance between i and any instruction in B.

LEVEL can be applied multiple times to different levels. Currently we apply it every four levels on Raw. The four levels correspond approximately to the minimum granularity of parallelism that Raw can profitably exploit given its communication cost.

Path propagation (PATHPROP) This pass selects high confidence instructions and propagates their convergent matrices along a path. The confidence threshold t is an input parameter. Let i_h be the selected confident instruction. The following propagates i_h along a downward path:

find $i \mid i \in \text{successor}(i_h); \text{confidence}(i) < \text{confidence}(i_h)$
while $(i \neq nil)$
  for each $(c,t), W_{i,t,c} \leftarrow 0.5W_{i,t,c} + 0.5W_{i_h,t,c}$
  find $i_n \mid i_n \in \text{successor}(i); \text{confidence}(i_n) < \text{confidence}(i_h)$
  $i \leftarrow i_n$

A similar function that visits predecessors propagates i_h along an upward path.

Emphasize critical path distance (EMPHCP) This pass attempts to help the convergence of the time information by emphasizing the level of each instruction. The level of an instruction is a good time approximation because it is when the instruction can be scheduled if a machine has infinite resources.

$$\text{for each } (i,c), W_{i,level(i),c} \leftarrow 1.2W_{i,level(i),c}$$

5 Results

We have implemented convergent scheduling in two systems: the Raw architecture [24] and the Chorus clustered VLIW infrastructure developed at MIT [20].

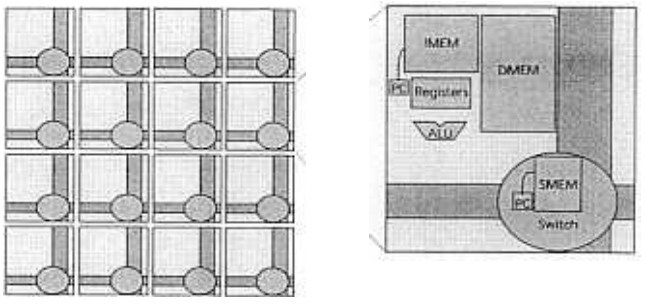

Figure 5. The Raw machine.

Experimental environment Figure 5 shows a picture of the Raw machine. The Raw machine comprises tiles organized in a two dimensional mesh. The actual Raw prototype has 16 tiles in a 4x4 mesh. Each tile has its own instruction memory, data memory, registers, processor pipeline, and ALUs. Its instruction set is based on the Mips R4000. The tiles are connected via point-to-point, mesh networks. In additional to a traditional, wormhole hole dynamic network, Raw has a programmable, compiler-controlled *static network* that can be used to route scalar values between the register file/ALUs on different tiles (for details, please refer to [14]). To reduce communication overhead, the static network ports are register-mapped. Latency on the static network is three cycles for two neighboring tiles; each additional hop takes one extra cycle of latency.

Rawcc, the Raw compiler, takes a sequential C or Fortran program and parallelizes it across Raw tiles. It is built on top of the Machsuif intermediate representation [18]. Rawcc divides each input program into one or more scheduling traces. For each trace, Rawcc constructs the data precedence graph and performs space-time scheduling on each graph. Then, it applies a traditional register allocator to the code on each tile.

The Chorus clustered VLIW system is a flexible compiler/simulator environment that can simulate many different configurations of clustered VLIW machines. We use it to simulate a clustered VLIW machine with four identical clusters. Each cluster has four function units: one integer ALU, one integer ALU/Memory, one floating point unit, and one transfer unit. Instruction latencies are based on the Mips R4000. The transfer unit moves values between register files on different clusters. It takes one cycle to copy a register value from one cluster to another. Memory addresses are interleaved across clusters for maximum parallelism. Memory operations can request remote data, with a penalty of one cycle.

The Chorus compiler shares with Rawcc the same

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

INITTIME	INITTIME
PLACEPROP	NOISE
LOAD	FIRST
PLACE	PATH
PATH	COMM
PATHPROP	PLACE
LEVEL	PLACEPROP
PATHPROP	COMM
COMM	EMPHCP
PATHPROP	
EMPHCP	
(a)	(b)

Table 1. Sequence of heuristics used by the convergent scheduler for (a) the Raw machine and (b) clustered VLIW.

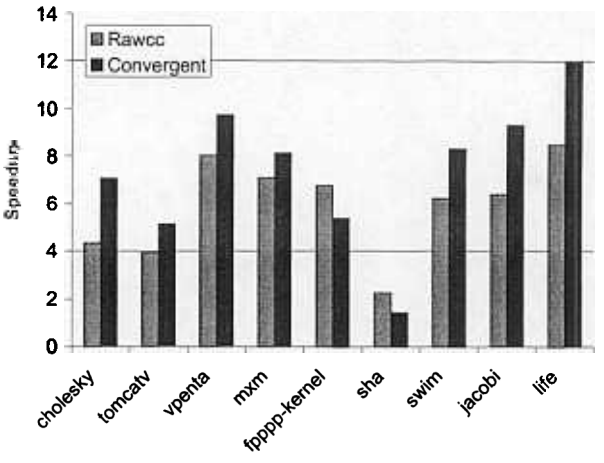

Figure 6. Performance comparisons between Rawcc and Convergent scheduling on a 16-tile Raw machine.

high level structure. Like Rawcc, it is implemented on top of Machsuif. It first performs space-time scheduling, followed by traditional single-cluster register allocation [8].

Both Rawcc and the Chorus compiler employ congruence transformation and analysis to increase and analyze the predictability of memory references [13]. This analysis creates preplaced memory reference instructions that must be placed on specific tiles or clusters. For dense matrix loops, the congruence pass usually unrolls the loops by the number of clusters or tiles. This unrolling also increases the size of the scheduling regions, so that no additional unrolling is necessary to expose parallelism.

In both compilers, when a value is live across multiple scheduling regions, its definitions and uses must be mapped to a consistent cluster. On Rawcc, this cluster is the cluster of the first definition/use encountered by the compiler; subsequent definitions and uses become preplaced instructions.[3] On Chorus, all values that are live across multiple scheduling regions are mapped to the first cluster.

Convergent schedulers Table 1 lists the heuristics used by the convergent scheduler for Raw and Chorus. The heuristics are run in the order given.

The convergent scheduler interfaces with the existing schedulers as follows. The output of the convergent scheduler is split into two parts:

1. A map describing the preferred partition, *i.e.,* an assignment for every instruction to a specific cluster.

2. The temporal assignment of each instruction.

[3]Rawcc does use SSA renaming to eliminate false dependences, which in turn reduces these preplacement constraints.

Both Chorus and Rawcc use the spatial assignments given by the convergent scheduler. Chorus uses the temporal assignments as priorities for the list scheduler. For Rawcc, however, the temporal assignments are computed independently by its own instruction scheduler.

Benchmarks Our sources of benchmarks include the Raw benchmark suite (jacobi, life) [1], Nasa7 of Spec92 (cholesky, vpenta, and mxm), and Spec95 (tomcatv, fpppp-kernel). Fpppp-kernel is the inner loop of fpppp from that consumes 50% of the run-time. Sha is an implementation of Secure Hash Algorithm. Fir is a FIR filter. Rbsorf is a Red Black SOR relaxation. Vvmul is a simple matrix multiplication. Yuv does RGB to YUV color conversion. Some problem sizes have been changed to cut down on simulation time, but they do not effect the results qualitatively.

Performance comparisons We compared our results with the baseline Rawcc and Chorus compilers. Table 2 compares the performance of convergent scheduling to Rawcc for two to 16 tiles. Figure 6 plots the same data for 16 tiles. Results show that convergent scheduling consistently outperforms baseline Rawcc for all tile configurations for most of the benchmarks, with an average improvement of 21% for 16 tiles.

Many of our benchmarks are dense matrix code with preplaced memory instructions from congruence analysis. For these benchmarks, convergent scheduling always outperforms baseline Rawcc. The reason is that convergent scheduling is able to actively take advantage

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

	Base				Convergent			
Benchmark/Tiles	2	4	8	16	2	4	8	16
cholesky	1.14	2.21	3.29	4.33	1.44	2.75	4.94	7.06
tomcatv	1.18	1.83	2.88	3.94	1.37	2.12	3.33	5.15
vpenta	1.86	2.85	4.58	8.03	1.96	3.23	5.82	9.71
mxm	1.77	2.40	3.78	7.09	1.89	2.54	4.04	7.77
fpppp	1.54	3.09	5.13	6.76	1.42	2.04	3.87	5.39
sha	1.11	2.05	1.96	2.29	1.05	1.33	1.51	1.45
swim	1.40	2.04	3.62	6.23	1.63	2.69	4.24	8.30
jacobi	1.33	2.43	4.13	6.39	1.40	2.74	4.92	9.30
life	1.65	3.02	5.56	8.48	1.76	3.35	6.34	11.97

Table 2. Rawcc speedup. Speedup is relative to performance on one tile.

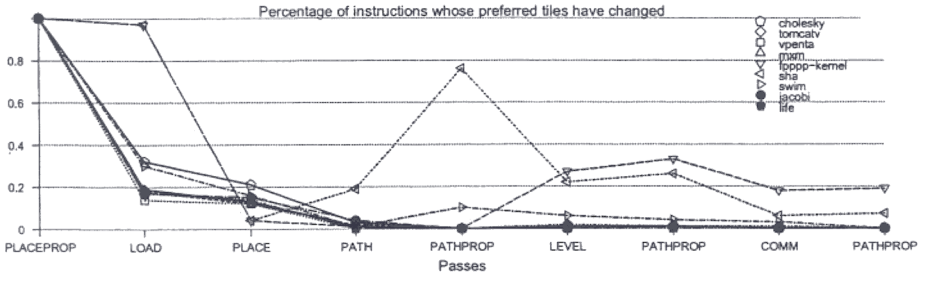

Figure 7. Convergence of spatial assignments on Raw.

of preplacement information to guide the placement of other instructions. This information turns out to lead to very good natural assignments of instructions.

For fpppp-kernel and sha, convergent scheduling performs worse than baseline Rawcc because preplaced instructions do not suggest many good assignments. Attaining good speedup on these benchmarks require finding and exploiting very fine-grained parallelism. Our level distribution pass has been less efficient in this regard than the clustering counterpart in Rawcc — we expect that integrating a clustering pass to convergent scheduling will address this problem.

Figure 7 shows the percentage of instructions whose preferred tiles are changed by each convergent pass on Raw. The plots measure static instruction counts, and they exclude passes that only modify temporal preferences. For benchmarks with useful preplacement information, the convergent scheduler is able to converge to good solutions quickly, by propagating the preplacement information and using the load balancing heuristic. In contrast, preplacement provides little useful information for fpppp-kernel and sha. These benchmarks thus require other critical paths, parallelism, and communication heuristics to converge to good assignments.

Figure 8 compares the performance of convergent scheduling to two existing assignment/scheduling techniques for clustered VLIW: UAS [23] and PCC [5]. We

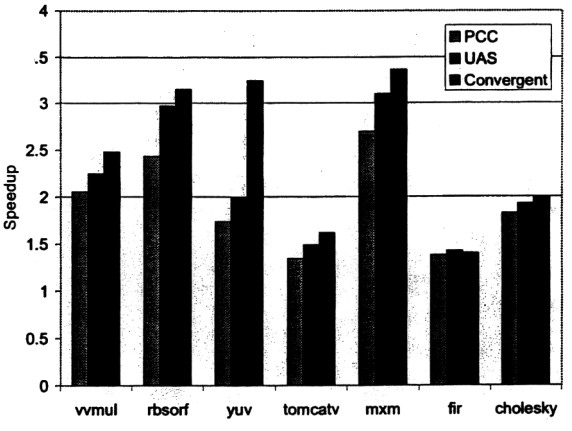

Figure 8. Performance comparisons between PCC, UAS, and Convergent scheduling on a four-clustered VLIW. Speedup is relative to a single-cluster machine.

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

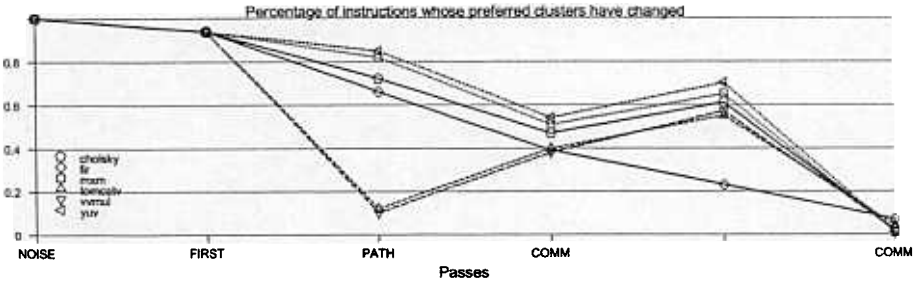

Figure 9. Convergence of spatial assignments on Chorus.

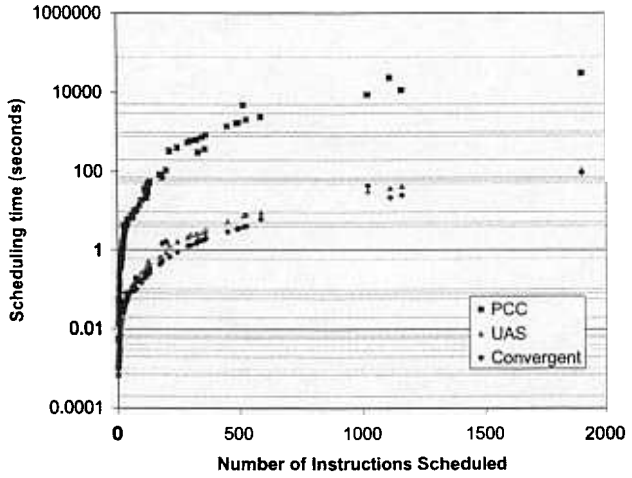

Figure 10. Comparison of compile-time vs input size for algorithms on Chorus.

augment each existing algorithm with preplacement information. For UAS, we modify the CPSC heuristic described in the original paper to give the highest priority to the home cluster of preplaced instructions. For PCC, the algorithm for estimating schedule lengths and communication costs properly accounts for preplacement information, by modeling the extra costs incurred by the clustered VLIW machine for a non-local memory access. Convergent scheduling outperforms UAS and PCC by 14% and 28%, respectively, on a four-clustered VLIW machine. Like in Raw, the convergent scheduler is able to use preplacement information to find good natural partitions for our dense matrix benchmarks. Figure 9 shows the percentage of static instructions whose preferred tiles are changed by each convergent pass on Chorus. Passes that only modify temporal preferences are excluded.

Compile-time scalability We examine the scalability of convergent scheduling. Scalability is important because there is an increasing need for instruction assignment and scheduling algorithms to handle larger and larger number of instructions. This need arises for two reasons: first, due to improvements in compiler representation techniques such as hyperblocks and treegions; and second, because a large scope is necessary to support the level of ILP provided by modern and future microprocessors.

Figure 10 compares the compile-time of convergent scheduling with that of UAS and PCC on Chorus. Both convergent scheduling and PCC use an independent list scheduler after instruction assignment — our measurements include time spent in the scheduler. The figure shows that convergent scheduling and UAS take about the same amount of time. They both scale considerably better than PCC. We note that PCC is highly sensitive to the number of components it initially divides the instructions into. Compile-time can be dramatically reduced if the number of components is kept small. However, we find that for our benchmarks, reducing the number of components also results in much poorer assignment quality.

6 Related work

Spatial architectures require cluster assignment, scheduling, and register allocation. We have provided a general framework that can perform all three tasks together (by adding preference maps for registers as well), but the focus of this paper is the application of convergent scheduling to cluster assignment.

Many compilers for spatial architectures address the three problems separately. Much research has focused on novel ways to do cluster assignment, coupled with traditional list scheduling and register allocation methods. The pioneer work in cluster as-

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

signment is BUG [6]. BUG is uses a two-phase algorithm. First, the algorithm traverses a dependence graph bottom-up to propagate information about preplaced instructions. Then, it traverses the graph top-down and greedily map each instruction to the cluster that can execute it earliest. The Multiflow compiler uses a variant of BUG [17], without the support for preplaced instructions. PCC is an iterative assignment approach based on partial components [5]. It builds partial components by visiting the data dependence graph bottom up, critical-path first. The maximum size of a component is capped by a parameter, ϕ_{th}. It uses simple heuristics to select a value for ϕ_{th} that balances the tradeoff between performance and compile-time, although the exact method is not explained. The components are initially assigned to clusters based on simple load balancing and communication criteria. The assignments are subsequently improved through iterative descent, by checking whether moving a sub-component to another cluster improves the schedule. Rawcc leverages techniques developed for multiprocessor task graph scheduling [14]. Assignment is performed in three steps: *clustering* groups together instructions that have little parallelism; *merging* reduces the number of clusters through merging; *placement* maps clusters to tiles. During placement, Rawcc also handles constraints from preplaced instructions.

In [3], the approach differs from the above approaches in the ordering of phases. It performs scheduling before assignment. The assignment phase uses a min-cut algorithm adapted from circuit partitioning that tries to minimize communication. This algorithm, however, does not directly attempt to optimize the execution length of input DAGs.

To avoid phase ordering problems, recent works have proposed combined solutions. Leupers describes an iterative combined approach to perform scheduling and partitioning on a VLIW DSP [16]. The approach is based on simulated annealing. UAS performs assignment and scheduling together by integrating assignment into a cycle-driven list scheduler [23]. CARS performs all three tasks — assignment, scheduling, and register allocation — in one step, by integrating both assignment and register allocation into a mostly cycle-driven list scheduler [11]. In all these approaches, however, every decision is irrevocable and final. In contrast, convergent scheduling provides a general framework that allows decisions to be postponed or reversed, and we have used it to address the phase ordering problem we found in cluster assignment when we have preplaced instructions.

Few assignment approaches are designed to take into account preplaced instructions. Of the above, only BUG and Rawcc directly support them.

Phase ordering issues on clustered architectures is a relatively new area; a more classical phase ordering problem occurs in scalar optimizations. Some approaches in those areas share similar goals and features with convergent scheduling. Cooper uses a genetic-algorithm solution to evolve the order of passes to optimize code size [4]. The approach finds good general solutions, and it performs even better when the evolution is applied independently on each benchmark. Lerner proposes an interesting interface to different passes based on *graph replacement* [15]. His approach enables independently designed dataflow passes to be composed and run together. The composed pass is able to achieve the precision of iterating independent passes, but without the compile-time cost of iteration.

7 Conclusion

This paper introduces convergent scheduling, a flexible framework for performing instruction assignment and scheduling. Its interface between passes are simple and expressive. The simplicity makes it easy for the compiler developer to handle new constraints or integrate new heuristics. It also allows passes to be executed multiple times or even iteratively, which helps alleviate phase ordering problems. The expressiveness allows passes to specify confidence about their decisions, thus avoiding poor irreversible decisions.

This paper presents evidence that convergent scheduling is a promising framework. We have implemented it on two spatial architectures, Raw and clustered VLIW, and demonstrated performance improvement of 21% and 14%, respectively. One reason for these performance gains comes from the ability of the convergent scheduler to support, in an integrated manner, both traditional constraints (parallelism, locality, and communication) and a new constraint (preplaced instructions).

References

[1] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal. The RAW Benchmark Suite: Computation Structures for General Purpose Computing. In *5th Symposium on FPGAs-Based Custom Computing Machines (FCCM)*, pages 134–143, 1997.

[2] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *26th International Symposium on Computer Architecture (ISCA)*, pages 4–15, 1999.

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 IEEE

[3] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs. In *25th International Symposium on Microarchitecture (MICRO)*, pages 292–300, 1992.

[4] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for Reduced Code Space using Genetic Algorithms. In *Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999.

[5] G. Desoli. Instruction Assignment for Clustered VLIW DSP Compilers: a New Approach. Technical Report HPL-98-13, Hewlett Packard Laboratories, 1998.

[6] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1986.

[7] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[8] L. George and A. W. Appel. Iterated Register Coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, 1996.

[9] W. A. Havanki, S. Banerjia, and T. M. Conte. Treegion Scheduling for Wide Issue Processors. In *4th International Symposium on High Performance Computer Architecture (HPCA)*, pages 266–276, 1998.

[10] W. Hwu, S. Mahlke, W. Chen, P. Chang, N. Warter, R. Bringmann, R. Ouellette, R. Hank, T. Kiyohara, G. Haab, J. Holm, and D. Lavery. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, 7(1):229–248, Jan 1993.

[11] K. Kailas, K. Ebcioglu, and A. K. Agrawala. CARS: A New Code Generation Framework for Clustered ILP Processors. In *7th International Symposium on High Performance Computer Architecture (HPCA)*, pages 133–143, 2001.

[12] H.-S. Kim and J. E. Smith. An Instruction Set and Microarchitecture for Instruction Level Distributed Processing. In *29th International Symposium on Computer Architecture (ISCA)*, pages 71–81, 2002.

[13] S. Larsen and S. Amarasinghe. Increasing and Detecting Memory Address Congruence. In *11th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2002.

[14] 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 *8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 46–57, 1998.

[15] S. Lerner, D. Grove, and C. Chambers. Composing Dataflow Analyses and Transformations. In *29th Symposium on Principles of Programming Languages (POPL)*, pages 270–282, 2002.

[16] R. Leupers. Instruction Scheduling for Clustered VLIW DSPs. In *9th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 291–300, 2000.

[17] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg. The Multiflow Trace Scheduling Compiler. In *Journal of Supercomputing*, pages 51–142, 1993.

[18] Machsuif. http://www.eecs.harvard.edu/hube.

[19] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective Compiler Support for Predicated Execution Using the Hyperblock. In *25th Annual International Symposium on Microarchitecture (MICRO)*, pages 45–54, 1992.

[20] D. Maze. Compilation Infrastructure for VLIW Machines. Master's thesis, Massachusetts Institute of Technology, September 2001.

[21] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen. Combining Register Allocation and Instruction Scheduling. Technical Report CS-TN-95-22, 1995.

[22] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. A Design Space Evaluation of Grid Processor Architectures. In *34th International Symposium on Microarchitecture (MICRO)*, pages 40–51, 2001.

[23] E. Ozer, S. Banerjia, and T. M. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures. In *31st International Symposium on Microarchitecture (MICRO)*, pages 308–315, 1998.

[24] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, J.-W. Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. S. M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, pages 25–35, March/April 2002.

Proceedings of the 35 th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)
1072-4451/02 $17.00 © 2002 **IEEE**