Convergent Scheduling: A Flexible and Extensible Scheduling Framework for Clustered VLIW Architectures

by

Diego Puppin

B.S., Università di Pisa, Italy (2000)
M.S., Scuola Normale Superiore di Pisa, Italy (2000)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

December 2002

Author .....
Department of Electrical Engineering and Computer Science
December 15, 2002

Certified by .....
Saman P. Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Convergent Scheduling: A Flexible and Extensible Scheduling Framework for Clustered VLIW Architectures

by

Diego Puppin

Submitted to the Department of Electrical Engineering and Computer Science
on December 15, 2002, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Convergent scheduling is a general instruction scheduling framework that simplifies and facilitates the application of a multitude of arbitrary constraints and scheduling heuristics required to schedule instructions for modern complex processors. A convergent scheduler is composed of independent phases, each implementing a heuristic that addresses a particular problem or constraint. The phases share a simple, common interface that allows to inquire and modify spatial and temporal preference for each instruction. With each heuristic independently applying its scheduling constraint in succession, the final result is a well formed instruction schedule that is able to satisfy most of the constraints.

We have implemented a set of different passes that addresses scheduling constraints such as partitioning, load balancing, communication bandwidth, and register pressure. By applying and hand-tuning these heuristics we are able to obtain an average increase in speedup on a 4-cluster clustered VLIW architecture of 28% when compared to Desoli's PCC algorithm [Des98], 14% when compared to UAS [OBC98], and a speedup of 21% over the existing space-time scheduler of the Raw processor [LBF+98].

Because phases can be applied multiple times and in any order, a convergent scheduler is presented with a vast number of legal phase orderings. We use machine-learning techniques to automatically search for good phase orderings, for three different VLIW architectures. The architecture-specific phase orderings yield speedups ranging from 12% to 95% over the baseline order. Furthermore, *cross validation* studies that we perform in this work show that our automatically generated orderings perform well beyond the benchmarks on which they were 'trained': benchmarks that were not in the training set are within 6% of the performance they would obtain had they been in the training set.

Thesis Supervisor: Saman P. Amarasinghe
Title: Associate Professor

Acknowledgments

This thesis expands and continues the work described in the paper *Convergent Scheduling*, presented at MICRO-35, Istanbul, November 2002.

I would like to thank the people whose help was important in the completion of this work. First, my advisor Saman Amarasinghe for his support and guidance during these two years. Then, David Maze, Sam Larsen, Michael Gordon and Mark Stephenson for the good group effort in developing the Chorus infrastructure. In particular, I would like to thank Mark for developing the initial genetic programming infrastructure, which was adapted for this work, and for his support to our paper on genetic programming.

I want to acknowledge the contribution given by Shane Swenson, with the initial work on soft scheduling, and Walter Lee, for his help with Raw compiler and for finalizing the Istanbul paper.

I would like to sincerely thank all the friends that made these two years a wonderful experience. This space is too small to remember all of them.

This work is dedicated to my family and to Silvia. Without their support and love I could not have made it.

Contents

List of Figures

List of Tables

Chapter 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 and VLIWs, the two primary issues are parallelism and register pressure. Code sequences that expose much 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. Figure 1-1 shows an example from [MPSR95] of such tradeoff.

On spatial architectures, instruction scheduling is even more complicated. Examples of spatial architectures include clustered VLIWs, Raw [WTS+97], Trips [NSBK01], and ILDPs [KS02]. Spatial architectures are architectures that distribute their computing resources and the register file. 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 1-2 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 desired, only one cycle is

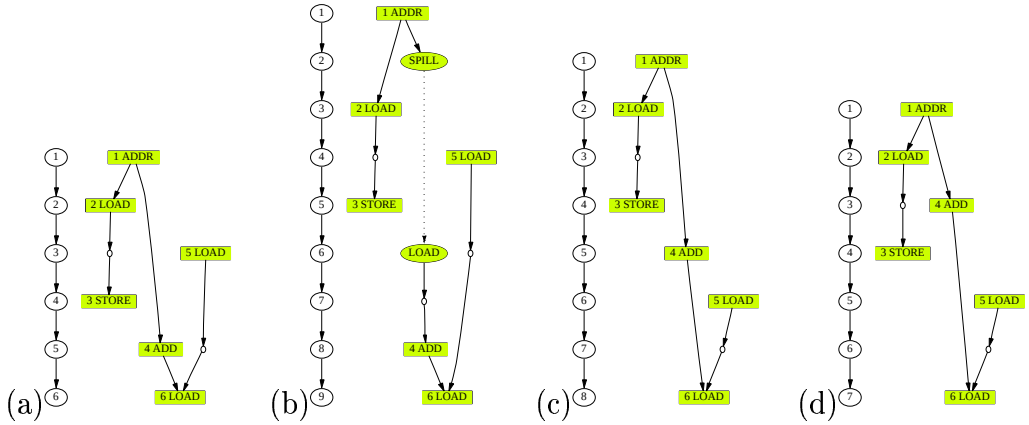


Figure 1-1: An example of tradeoff between aggressive scheduling and register pressure. Rectangles are instructions; edges between rectangles represent data dependences, with circles on them representing delays due to instruction latency. The circles on the left represent the time axis. Consider a single-issue machine with two registers and two-cycle loads. Figure (a) shows an aggressive schedule that attempts to overlap the load latencies. After cycle three, there are three live ranges, so one value must be spilled. The spilling leads to the code sequence in (b), which takes nine cycles. If instead the scheduler tries to minimize register pressure, we end up with schedule (c), which still takes eight cycles. The optimal schedule, in (d), takes only seven cycles, and it exhibits a careful tradeoff between aggressive scheduling and register pressure minimization.

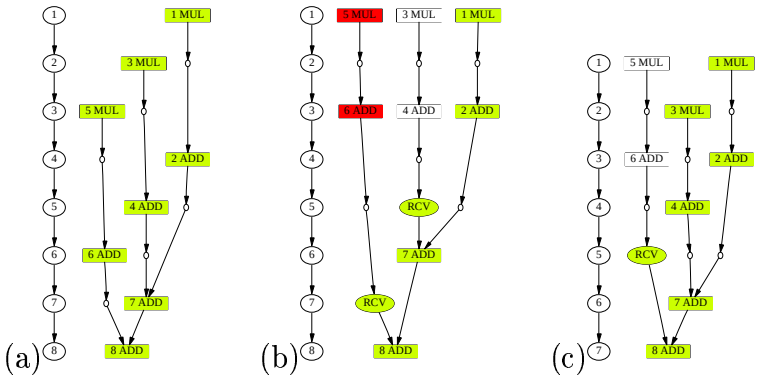


Figure 1-2: An example of tradeoff between parallelism and locality on spatial architectures. Each node color represents a different cluster. Consider an architecture with three clusters, each with one functional unit and three registers, where communication takes one cycle of latency due to the receive 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.

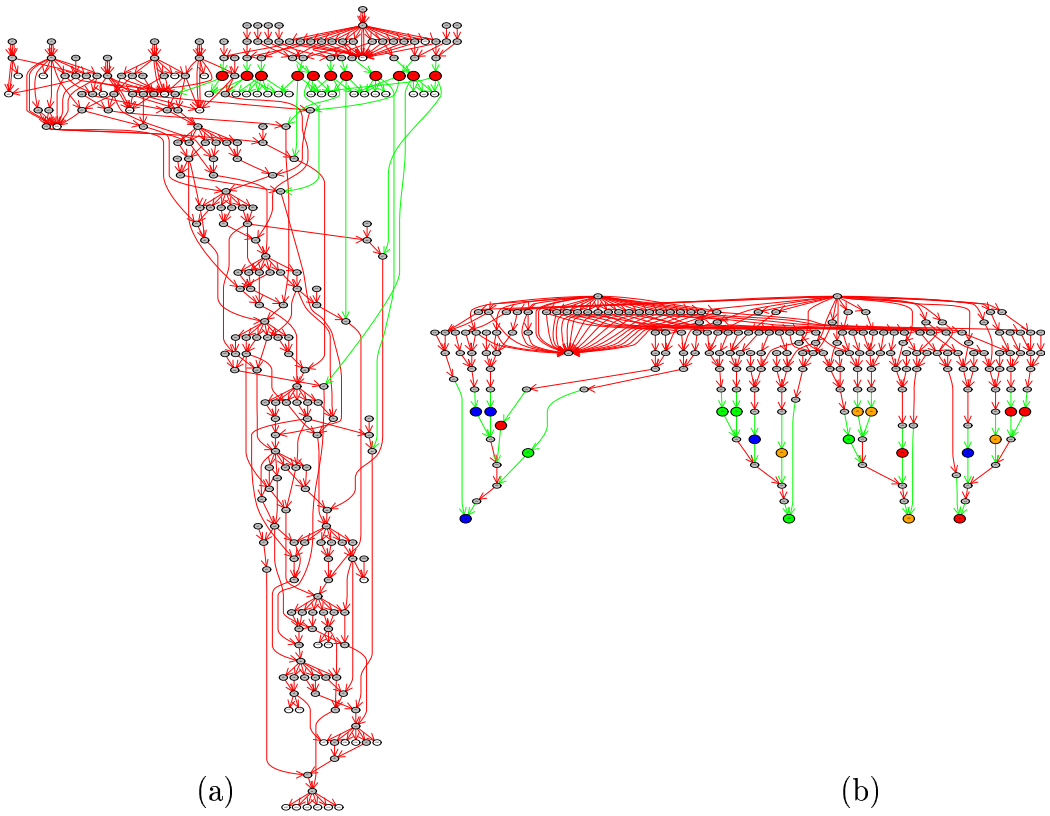


Figure 1-3: Two examples of data dependence graphs. (a) is a typical example from a non-numeric program, (b) is common in unrolled numerical loops

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 may have specific spatial requirements. For example, these requirements may arise from the need to access specific spatial resources, such as a specific memory bank [BLAA99]. A good scheduler must be sensitive to these constraints in order to generate a good schedule.

A scheduler also faces difficulties because different heuristics work well for different types of graphs. Figure 1-3 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 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 [MPSR95]. However, extending such frameworks to support additional spatial constraints is difficult – no such extension exists today.

Also, commercial processors can share the same Instruction-Set Architecture, but with very different internal organizations. The cost of targeting with effectiveness the new architecture grows with the faster turn-over of processor generations. A system to address this, in an automatic manner, is strongly needed.

This thesis presents *convergent scheduling*, a general scheduling framework that makes it easy to specify arbitrary constraints and scheduling heuristics. Figure 1-4 illustrates this framework. A convergent scheduler is composed of independent 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 and output to each phase is a collection of spatial and temporal preferences of instruc-

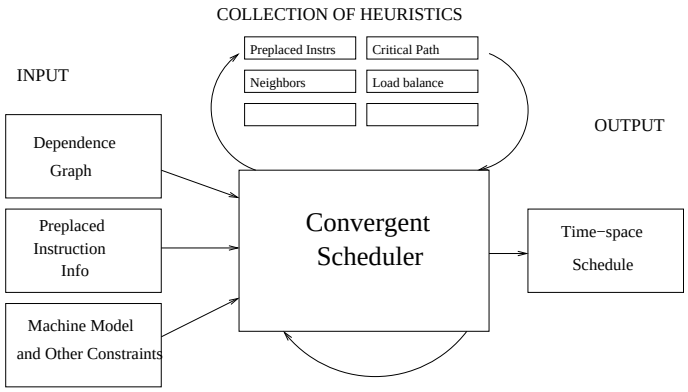


Figure 1-4: The convergent schedule infrastructure.

tions. A phase operates by modifying these data. As the scheduler applies the phases in succession, the preference distribution will converge 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 three-tuple to a weight.

In our first work with convergent scheduling, we tediously hand-tuned the phase order. While the sequence works well for the processors we initially explored, it does not generally apply to new architectural configurations. As we add new phases to our scheduler to address next generation architectural features, hand-tuning the sequence of passes becomes even harder.

To complicate matters, architectures evolve quickly. Even though a processor family may share the same programming interface (ISA), the internal organization of the processors can differ dramatically (*e.g.*, number of registers, functional units, etc.). It is the compiler's task to address the architectural features efficiently, by determining a schedule that matches the constraints. Time-to-market pressures make it extremely difficult to effectively target new architectures.

This thesis uses machine learning techniques to automatically find good phase orderings for a convergent scheduler. We show how our system can automatically discover architecture-specific phase orders. Because different parallel architectures have unique scheduling needs, the speedups our system is able to obtain by creating architecture-specific phase orderings is impressive. Equally impressive is the ease with

which it finds effective sequences.

Using a modestly sized cluster of workstations, our system is able to quickly find good convergent scheduling sequences. In less than two days, it discovers sequences that produce speedup ranging from 12% to 95% over previous work. Furthermore, by varying architectural parameters and rerunning the experiment, we show that different architectures indeed have special compilation requirements. The learning algorithm catered a sequence of passes to each of the three architectures on which we tested it.

The main contributions of this thesis are:

- a novel approach to address the combined problems of partitioning, scheduling, and register pressure,
- the formulation of a set of powerful heuristics to address very general constraints and some architecture-specific issues,
- a demonstration of the effectiveness of convergent scheduling, which is able to surpass more complex combined solutions,
- the use of machine learning to adapt convergent scheduling to a new architecture.

The rest of this thesis is organized as follows. Chapter 2 introduces convergent scheduling and uses an example to illustrates how it works. Chapter 3 gives more detail about infrastructure and implementation. Chapter 4 discusses how the system can be adapted to different architectures by means of genetic programming. Chapter 5 illustrate our compiler infrastructure and the schedulers we used for our experimental comparisons. Chapter 6 presents results for a clustered VLIW architecture and for the Raw processor. Chapter 7 describes the framework and the results we reached when we adapted our system to different VLIW architectures. Chapter 8 provides related work. Chapter 9 highlights future work and concludes.

Chapter 2

Convergent scheduling

In this chapter, we introduce convergent scheduling by giving an example of its work on a basic block from fpppp. With this, we show the peculiar features of the system and how it avoids some of the problems typical of more traditional compilers.

In the convergent scheduling framework, passes communicate their choices as changes in the relative preferences of different schedules. A pass works by manipulating the *weight* for a specific instruction to be scheduled at a specific cycle, in a specific cluster.[1] At the end of the algorithm, every instruction will be scheduled in the space-time slot with the heighest weight, which we call the *preferred* slot.

Different heuristics work to improve the schedule in different ways. The *critical path (CP) strengthening* heuristic, for example, expresses a preference to keep all the instructions in the CP together in the same cluster, by determining the best cluster for this, and by increasing the preference (weights) for those instructions to be scheduled there. The *communication minimization* heuristic tries to keep dependent instructions (neighbors) in the same cluster, by computing for every instruction where most of its neighbors are, and then by increasing the preference for that cluster. The *preplacement* heuristic considers the congruence information, as defined in [LA02], to exploit the memory parallelism while preserving locality. If the memory is banked and every bank is local to a cluster, this heuristic increases the preference to keep memory

[1] In the rest of this thesis, we will use interchageably the terms *phases* and *passes*, *tile* and *cluster*, and *cycle* and *time slot*.

instructions in the cluster where most of the dynamic instances of the instruction are local. The *load balance* heuristic reduces the preferences on the most loaded cluster, and increases them on the least loaded one. Other passes will be introduced in section 3.3.

Figure 2-1 shows how convergent scheduling operates on a small code sequence from fpppp. In this simple example, we will focus only on the heuristics that address space allocation. Figure 2-1(a) shows the data dependence graph of the code sequence. Each node is an instruction, and each edge represents a dependence between instructions. Triangular nodes represent preplaced instructions. For simplicity, the example only illustrates space scheduling, not the combined space and time scheduling. Each of the figures 2-1(b–g) is a cluster preference map. A row represents an instruction. The row numbers corresponds 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.

Initially, the weights are evenly distributed, as shown in (b). We apply the *noise introduction* heuristic to break symmetry, resulting in (c). This heuristic helps increase parallelism by distributing instructions to different clusters. Then, we run *critical path (CP) strengthening*, which increases the weight of the instructions in the CP (i.e. instructions 23, 25, 26, etc.) in the first cluster (d). Then we run the *communication minimization* and the *load balance* 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 the third cluster).

Next, we run a *preplace biasing* pass that utilizes information about preplaced nodes. The result is shown in (f). This pass causes a lot of disturbances: preplaced instructions strongly attract their neighbors to the same cluster. Observe how the group 19–22 is attracted to the last cluster. Finally we run *communication minimization* another time. The final schedule is shown in (g).

The schedule is very effective because it reaches a good trade-off over conflicting opportunities: parallelism is exploited, but keeping in consideration the memory

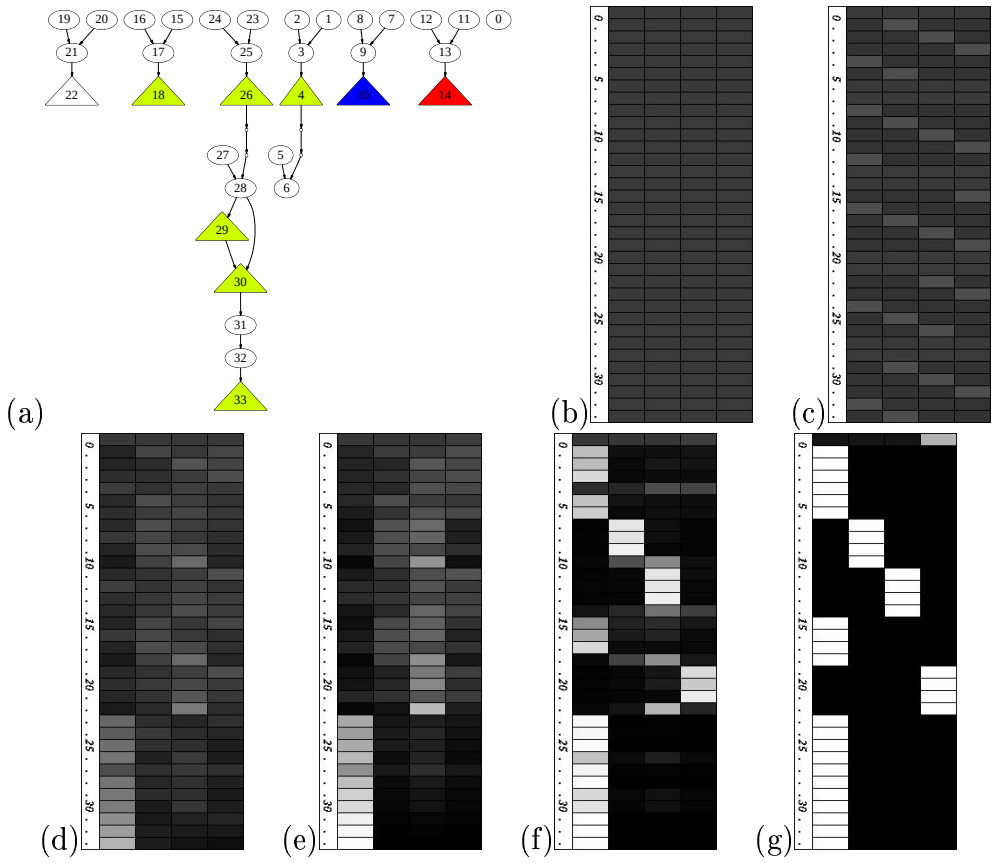


Figure 2-1: Convergence scheduling operates on a code sequence in from fpppp. Rows represent the different instructions i in the block, columns are the (four) clusters c in the architecture. Every slot (i, c) represents the weight of i to be scheduled on cluster c. The brighter the color, the higher the weight. The dependence graph relative to the block is shown. Triangular nodes are preplaced, with different shades corresponding to different clusters. Rows are numbered according to the node numbers in the graph.

layout and instruction preplacement; critical path is kept together so to minimize delays due to communication; independent subcomponents of the graph are moved to unused tiles.

Convergent scheduling has the following features:

1. 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 a randomizer to (b), many 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 move 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 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 instruction.

7. The very clean design allows to easily re-order, add or remove passes from the compiler. This gave us the opportunity to design a system that *evolves* the sequence of passes in order to adapt to the underlying architecture.

Chapter 3

Implementation

This chapter describes in detail our implementation of convergent scheduling: the way we implemented *preferences*; the driver infrastructure; our heuristics; the set of metrics we used in our adaption experiments, and how we combined them into boolean tests.

3.1 Preferences

Convergent scheduling operates on individual scheduling units, which may be basic blocks, traces, superblocks, or hyperblocks. It stores preferences in a three dimensional matrix $W_{i,t,c}$, 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 cycles as the critical-path length (CPL), with the goal of finding an optimal schedule that fits the CPL. Even when this is not possible, our framework allows us to return rich information that will allow to build an optimal schedule in this case too. This is due to the fact that we compute a preference map that is much richer than a simple time schedule for the instructions. For example, our system can in fact verify the presence of phases in the schedule, which can be highlighted and brought to the attention of the list scheduler: the system can identify feasible and unfeasible time-slots for each instruction in the final schedule, more than just giving a time schedule.

Initially, all the weights are distributed evenly. Each pass examines the depen-

dence 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 avoid any changes, or change weights only slightly. It can be the case that following passes will pick up the hint to change the preferred schedule.

If i spans over instructions, t over time-slots, c over clusters, we have:

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

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

We define:[1]

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

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

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

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

Preferred values are those that maximize the sum of the preferences over time and clusters. The *preferred schedule* is the one obtained by assigning every instruction to its preferred space-time slot. The runner-up cluster is the second best. The confidence is given by the ratio of the preference for the preferred and the runner-up cluster.

Some basic operations are available on the weights:

[1] 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.

- the weight of a specific instruction to be scheduled in a given space-time slot can be increased or decreased by a constant, and multiplied or reduced by a factor;

- the system keeps track of the sums over rows and columns, and of their maximum, so it can quickly, in time $O(1)$, determine the sum of weights over time and clusters, and then the preferred space-time slot;

- the preferences can be normalized to guarantee our invariants; the normalization simply performs:

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

3.2 Configurable Driver

Convergent scheduling works on the SUIF representation of scheduling units (*blocks*, in the following).[2] The chosen sequence of passes is described by a *genome*, which allows conditional execution of certain passes.[3]

The convergent scheduler creates a preference matrix for the block, runs the chosen passes, updates the preferences according to each pass, determines a preferred time-space schedule (see table 3.1). This information is then passed to a list scheduler, and then to the register allocator. As explained, convergent scheduling determines the optimal schedule considering the needs of the list scheduler and the register allocator, so the performance of these two parts is predictable.

In chapter 5, we will give more details about how we integrated convergent scheduling with our compiler infrastructure.

[2]Basic blocks for the Chorus clustered VLIW system, and single-entry single-exit regions, flattened with predication, for RAW.

[3]For the moment, let just think at the genome as a simple sequence of passes. For more detail, see section 3.4.

function *convergent_scheduling*:

input: CFG describing the scheduling unit,
genome describing the sequence of passes
output: instruction partition *partition()*,
instruction priorities *priority()*

build the empty preference matrix
for every pass p in the genome:
apply the pass on the preference matrix
update and normalize the matrix

for every instruction i:
partition(i) = preferred_cluster(i)
priority(i) = preferred_time(i)

Table 3.1: Pseudo-code for the driver of convergent scheduling

3.3 Collection of Heuristics

The previous sections introduced the driver and the common data structure used by the heuristics of convergent scheduling. This section introduces the rich collection of heuristics we have implemented so far. 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.

3.3.1 Time Heuristics

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 length 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.

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

When we normalize, the weight for the suitable time-slots will increase suitably. With some effort, this pass can anyway be expressed in a close form that keeps the invariants. This is true also for the other passes listed here. We will give here implementations exploiting normalization instead of close expressions when they are simpler.

$$S_i \stackrel{\text{def}}{=} \sum_{c, l_p \le t \le CPL - l_s} W_{i,t,c}$$

$$\text{for each } i, c, t, W_{i,t,c} \leftarrow \begin{cases} W_{i,c,t}/S_i & \text{if } l_p \le t \le CPL - l_s \\ 0 & \text{otherwise} \end{cases}$$

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

Dependence enforcement (DEP)

Sometimes, a pass can change weights so that the preferred time of an instruction i is earlier than that of another instruction j that creates a result need by i ($j \rightarrow i$). In this case, to help the convergence, we reduce the weights for i to be scheduled before the preferred time t_j of j (plus its latency).

$$\begin{array}{l} \text{for each } i, \text{for each } j \in \text{predecessors}(i), \\ \quad \text{if } t_i < t_j \\ \quad\quad \text{for each } c, 0 \le t < t_j + \text{latency}(j), \\ \quad\quad\quad W_{i,t,c} \leftarrow 0.3 W_{i,t,c} \end{array}$$

Functional units (FUNC)

This pass considers the utilization of the functional units for every time slot. If a time-space slot is overburdened by a large number of instructions, their weights for that slot are reduced. At the moment, this heuristic assumes that there is just one general purpose functional unit available per slot, but this can be easily extended to consider the number and type of functional units in the machine.

$$\text{load}(c,t) \stackrel{\text{def}}{=} \sum_i W_{i,t,c}$$

$$\text{for each } i,t,W_{i,t,c} \leftarrow \begin{cases} 0.9W_{i,t,c} & \text{if load(c,t)} > 1 \\ W_{i,t,c} & \text{otherwise} \end{cases}$$

Emphasize critical path distance (EMPHCP)

This pass attempts to help the convergence of information about time by emphasizing the level of each instruction. Given instruction i, we define $level(i)$ to be its distance from the furthest root. 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.

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

3.3.2 Placement and Critical Path

Push to first cluster (FIRST)

In the clustered VLIW infrastructure we used, an invariant is that all the data are available in the first cluster at the beginning of every block. For this architecture, we want to give advantage to a schedule that utilizes the first cluster, where data are already available, more than the other clusters, where copies can be needed. In our framework this is easily expressed.

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

Preplacement (PLACE)

This pass increases the weight for preplaced instructions in their home cluster.

In our experiments with clustered architectures, we verified the importance of loop unrolling and of exploiting correctly the local memory. An access to a local memory location is faster and more efficient than an access to remote memory (in another cluster). That is why we want to place a static memory operation in the cluster to which most of its dynamic instances refer. This is called the memory operation's *home cluster*. In both Raw and the Chorus clustered VLIW architecture, the home cluster is determined using congruence analysis [LA02].

$$\text{for each } i,t,c, \text{if } i \text{ preplaced } W_{i,t,c} \leftarrow \begin{cases} W_{i,t,c} & \text{if } c \text{ is the home cluster} \\ 0 & \text{otherwise} \end{cases}$$

Preplacement propagation (PLACEPROP)

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

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

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 (preplacement) 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.

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

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:

given i_h
for each $i \in \text{successor}(i_h) : \text{confidence}(i) < \text{confidence}(i_h)$,
  for each $(c, t), W_{i,t,c} \leftarrow 0.5W_{i,t,c} + 0.5W_{i_h,t,c}$

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

Create clusters (CLUSTER)

PLACE was found to be a very strong and effective heuristics. Nonetheless, if the program to be compiled does not feature natural preplacement, we try to build clusters of nodes which should stay together, and we distribute them across tiles trying to improve parallelism. This heuristic is rather complex, we will try to give here a high-level overview.

1. identify the candidate clusters, using Desoli's `partial_components` algorithm (see [Des98]), with threshold equal to the size of the graph divided by the number of tiles,
2. for every instruction cluster:
   (a) if some instructions are preplaced, skip to the next;
   (b) otherwise, assign it to the next cluster (round-robin), by marking its center as placed in that cluster;

3. run a modified PLACE which keeps into account marked instructions as if they were preplaced.

Heuristically, we consider the center of the cluster as the instruction that was added to the cluster as the $N/2$-th, if the size of the cluster is N. In the future, we are planning to use the DSC algorithm to build cluster [GY94].

3.3.3 Communication and Load Balancing

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 the sum to skew weights in the correct direction.

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

We wrote a version of this pass that considers *grand-parents* and *grand-children*. We usually run it together with COMM.

Parallelism for successors (SUCC)

This is an example of an architecture-specific pass. In some configurations of our clustered VLIW infrastructure, data passed from one cluster to another can be snooped by other clusters. This way, we can easily implement a broadcast operation. We exploit this fact by scattering the successors of any instructions to the various clusters if some successor is already placed in a different cluster (a communication is already needed). This is going to improve parallelism and reduce register pressure without requiring more communication.

$$
\begin{array}{l}
\text{for each } i, \text{if } \sharp\{\text{successors}(i)\} > 2 \\
\quad \text{and } \exists n_1 : (\text{preferred\_cluster}(\text{n}_1) \neq \text{preferred\_cluster(i)}) \\
\quad\quad \text{for each } i \in \{\text{successors}(i)\}, t, \\
\quad\quad\quad r \leftarrow \text{randomly chosen cluster} \\
\quad\quad\quad W_{i,t,r} \leftarrow 2\ W_{i,t,r}
\end{array}
$$

Load balance(LOAD)

This pass looks for the most loaded cluster, and reduces the weight of instructions to be scheduled there (so increasing the weight for the other slots).

$$
\begin{array}{l}
\text{max cluster} \stackrel{\text{def}}{=} \arg\max \{c : \sum_i \sum_t W_{i,t,c}\} \\
\text{for each } i, t, W_{i,t,c} \leftarrow \begin{cases} 0.9 W_{i,t,c} & \text{if } c = \text{max cluster} \\ 1.1 W_{i,t,c} & \text{otherwise} \end{cases}
\end{array}
$$

Level distribute (LEVEL)

This pass distributes instructions at the same *level* across clusters. 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 for each cluster c contains instructions whose preferred cluster is c, and whose confidence is greater than a threshold, here equal to 2. The algorithm is described in table 3.2.

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

LevelDistribute: input int l, int g
  I_l = Instruction i : level(i) = l
  for each c,
    $I_l = I_l - B_c$
  $I_g = \{i \in I_l : \text{distance}(i, \text{find\_closest\_bin}(i)) > g\}$
  while $I_l \neq \phi$
    $B = \text{round\_robin\_next\_bin}()$
    $i_{closest} = \arg\max\{i \in I_g : \text{distance}(i, B)\}$
    $B = B \cup i_{closest}$
    $I_l = I_l - i_{closest}$
    Update I_g
  for each c,
    for each $i \in B_c$,
      for each $t, W_{i,t,c} \leftarrow 10W_{i,t,c}$

Table 3.2: The algorithm LevelDistribute.

granularity of parallelism that Raw can profitably exploit given its communication cost.

3.3.4 Register allocation

Break edges (EDGES)

This pass tries to compute the number of live ranges at a specific time t and cluster c. We approximate this number with the number of edges the head of which is scheduled before t and the tail after t.This clearly does not take into account the fact that two or more edges could be referring to the same variable, but it is a good approximation before register allocation.

In the convergent scheduling framework, we have to consider the weight associated to a specific time-space schedule. For every edge, $e_{t,c}$ is defined as the product of the sum of the weights for the head to be scheduled before t and for the tail to be scheduled after t. If the total weighted number of edges is large than N the number of registers in the architecture, we reduce the weights on t, with the goal of breaking the edges, i.e. scheduling the head after t or the tail before t.

$$\text{head}(i,T,c) \stackrel{\text{def}}{=} \sum_{i,t<T,c} W_{i,t,c}$$
$$\text{tail}(i,T,c) \stackrel{\text{def}}{=} \sum_{i,t>T,c} W_{i,t,c}$$
for each c,t,
  if $\left(\sum_{edges(a,b)} \text{head}(a,t,c) * \text{tail}(b,t,c)\right) > N$
    for each $i, W_{i,c,t} \leftarrow 0.8 W_{i,c,t}$

Reduce parallelism (SEQUENTIAL)

This pass tries to keep together (time- and space-wise) instructions that follow each other in the original set of instructions and that are dependent from each other. We do so by increasing the weight of the instruction following i to be in the same preferred cluster c_i, and in the next time slots of the preferred time t_i, that is $t_i + 1$.

This is going to minimize the number of temporary values with long life span. This clearly has an effect of performance, because it reduces parallelism, and so it requires careful balancing with other heuristics.

$$\text{next}(i) \stackrel{\text{def}}{=} \text{first dependent instruction following } i \text{ in the block}$$
$$\text{for each } i, W_{next(i),t_i,c_i} \leftarrow 1.2 W_{next(i),t_i+1,c_1}$$

3.3.5 Miscellaneous

Noise introduction (NOISE)

This pass introduces some noise in the weight distribution, so to break symmetry for subsequent choices. This is important, for instance, in order to have a good allocation of the critical path. After the PATH pass has identified the critical path, the presence of noise will help to perform an unbiased choice of the target cluster. This can defend the system from worst-case scenarios.

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

Assignment strengthening (BEST)

This pass simply boosts the preference for the preferred slot for every instruction. This is useful as a last pass, but also as a middle pass, in order to strengthen the preferences till the point. If t_i and c_i are again the preferred time and cluster for i:

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

3.4 Collection of Metrics

Along with the passes, we designed a series of metrics, used to measure and determine the current status of the schedule, and the shape of the block being analyzed. Our system is going to exploit this information in order to choose the best passes to run, and the strength used by them: the driver for convergent scheduling can execute one or more passes conditionally, according to the results returned by the metrics.

3.4.1 Graph size

This returns the number of instructions in the block, and can be used to build more complex expressions.

3.4.2 Unplaced

Unplaced returns the number of instructions that are further than a distance of 4 from a preplaced instructions, or that are close (within distance 4) to two (or more) instructions preplaced to different clusters. If *unplaced* is high, it means that communication will be needed to move data across clusters, because instruction will not naturally partition into clusters. In this case, COMM will be needed to minimize the delays due to communication

3.4.3 CPL

This returns the critical path length of the block.

3.4.4 Imbalance

Imbalance is a measure of the load-balance in the current schedule: it returns the maximum difference of load between any two clusters. If a block is particularly imbalanced, the LOAD pass can effectively improve the overall scheduling. Also, a block can be imbalanced because of the presence of the very long critical path that dominates the schedule. In this case, discriminating on the number or the size of critical paths can help take further decisions on the schedule.

$$\text{load}(c) \stackrel{\text{def}}{=} \sum_{i,t} W_{i,t,c}$$
$$\text{imbalance} \stackrel{\text{def}}{=} \max\{i, j : |\text{load}(i) - \text{load}(j)|\}$$

3.4.5 Number of CPs

This function returns the number of independent paths the length of which equals the critical path (CP) length. This is used to determine the parallelism present within the analyzed scheduling unit. The presence of multiple critical path can be caused by unrolling or by intrinsic parallelism in the program. Independent critical path can effectively be assigned to different cluster with no performance penalty.

determine one critical path CP
number_of_CPs = 0
$CPL \leftarrow length(CP)$
mark every $i \in CP$ as used
for each $r \in roots$,
  find the longest length path P from r
  if all $i \in P$ are not marked as used
    number_of_CP +=1
    mark every $i \in P$ as used
return number_of_CPS

3.5 Boolean tests

These metrics have been combined into simple boolean tests that can be used by the driver to perform choices on the schedule. We are planning to extend our genome syntax to include arithmetic expressions and comparison. In such an infrastructure, this simple boolean tests will not be necessary anymore.

Is imbalanced is true if *imbalance* is larger than $1/numcluster$.

Is fat is true if *number of CPs* is larger than the number of tiles.

Is within CPL is true if the number of instructions in the block is smaller than the number of tiles times the CPL.

Is placement bad is true if the number of *unplaced* instructions is more than half the number of instructions in the block.

Chapter 4

Adapting Convergent Scheduling by means of Genetic Programming

From one generation to the next, architectures in the same processor family may have extremely different internal organizations. The Intel Pentium® family of processors is a case in point. Even though the ISA has remained largely the same, the internal organization of the Pentium 4 is drastically different from that of the baseline Pentium.

To help designers keep up with market pressures, it is necessary to automate as much of the design process as possible. In our initial work with convergent scheduling, we tediously hand-tuned the sequence of passes. While the sequence works well for the processors we explored in our previous work, it does not generally apply to new architectural configurations. Different parallel architectures necessarily emphasize different grains of computation, and thus have unique compilation needs.

We therefore developed a tool to automatically customize our convergent scheduler to any given architecture. The tool generates a sequence of phases from those described in section 3.3. This chapter describes genetic programming (GP), the machine-learning technique that our tool uses.

Genetic programming (GP) is one of many machine-learning techniques. Like other learning algorithms, GP is based on the idea of evolution: a population of individuals are set to compete against each other in a specific task. The fittest

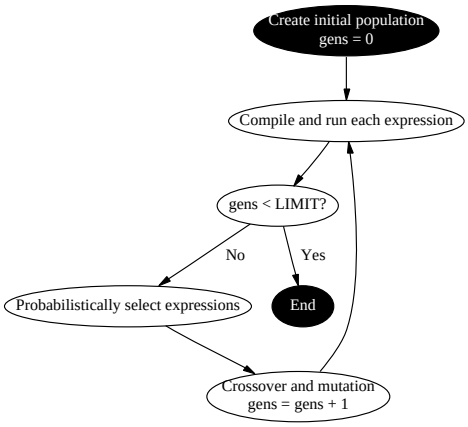


Figure 4-1: Flow of genetic programming. Genetic programming (GP) initially creates a population of expressions. Each expression is then assigned a fitness, which is a measure of how well it satisfies the end goal. In our case, fitness is proportional to the execution time of the compiled application(s). Until some user-defined cap on the number of generations is reached, the algorithm probabilistically chooses the best expressions for mating and continues. To guard against stagnation, some expressions undergo mutation.

ones are able to reproduce and generate off-springs, which will carry on the fight for survival. As in the Darwinian representation of evolutions, the least fit creatures will not replicate their *genome*, which will disappear from the population. GP models sexual reproduction, by having cross-over of the genomes of the fittest individuals, and allows random mutations to introduce new genomes in the population.

GP has a set of features that makes it particularly fit to our task: it is suited to explore high-dimensional spaces; it is highly scalable, highly parallel and can run effectively on a distributed computer farm; it presents solution that are readable to humans, compared with other algorithms (e.g. neural networks) where the solution is embedded in a very complex state space.

In the general GP framework, individuals are represented as parse trees [Koz92]. They are interpreted as a conditional sequence of passes: a grammar for our expressions is in table 4.1.

4.1 Harness

The harness used in our experimentation is adapted from the work of [SAMO02], the structure of which is described in figure 4-1.

The system controls the evolution of a set of expressions challenged with the compiling of a set of benchmarks. The initial, given seed expression is used to compile and run all benchmarks. The performance (elapsed number of cycles needed to complete the execution) is stored as a reference for each benchmark.

An initial set of individuals is created randomly to populate the world. Each genome is tested against the benchmarks. The fitness is determined by computing the average of the speed-up on single benchmarks, compared with the baseline running time. Individuals with the best performance are considered the fittest. As a secondary criterion, we favors individuals with shorter genomes, as in [Koz92, p. 109]. Shorter sequences offer more insight in the problem under analysis, are easier to read and understand, and lead to a shorter and faster compiling.

The fittest individuals are chosen to mate and reproduce. Sexual reproduction is an important part of GP: sub-expressions from strong individuals are swapped at the moment of reproduction (cross-over). This contributes to add variety to the population, and to reward strong genomes. Our harness uses a strategy called *tournament selection*, to choose the individuals that will reproduce. The tournament randomly chooses a set of n individuals, and then choose the best of them for reproduction (see [Koz92]).

In our framework, the reproduction by crossover chooses two subtrees from each of the parents, which are swapped to create two off-springs. As in [KH99], our harness uses *depth-fair crossover*, which gives fair opportunity to all the levels in the tree: a naive approach would choose leaves more often (in a binary tree, 50% of nodes are leaves).

After the off-springs are generated, a subset of them is subject to random mutations, which increase further the diversity in the genetic pool. The process iterates till the number of generations reaches a chosen number.

⟨*sexpr*⟩ ::= ('sequence' ⟨*sexpr*⟩ ⟨*sexpr*⟩)
| ('if' ⟨*variable*⟩ ⟨*sexpr*⟩ ⟨*sexpr*⟩)
| (⟨*pass*⟩)

⟨*variable*⟩ ::= #1 - Is imbalanced
| #2 - Is fat
| #3 - Is within CPL
| #4 - Is placement bad

⟨*pass*⟩ ::= 'PATH' | 'COMM' | 'NOISE' | 'INITTIME'
| 'SUCC' | 'LOAD' | 'EDGES' | 'DEP'
| 'BEST' | 'FUNC' | 'PLACE' | 'SEQUENTIAL'
| 'FIRST' | 'CLUSTER' | 'EMPHCP'

Table 4.1: Grammar for genome s-expressions.

4.2 Grammar

In our genetic programming framework, the sequence of passes is coded as a LISP s-expression. This allowed us to easily take advantage of the rich body of results about genetic programming and evolution of s-expressions. We knew that a good compiler could improve incrementally by adding or removing passes, or by switching the order of two of them. All these operations are performed during the evolutionary process by GP frameworks.

<variable> returns the value computed by our tests on the graph and the current schedule (see section 3.4). For simplicity, in the following, we will refer to the sequence `(SEQ (PassA) (PassB))` simply as `(PassA) (PassB)`: when no variables are used, genomes reduces to a linear sequence of passes.

This system is able to retarget convergent scheduling to new architectures, effectively. Adaptation can be run overnight on a clusters of workstations, to find a genome that produces good schedules for the target architecture. In chapter 7, we describe the details of our experiments and results.

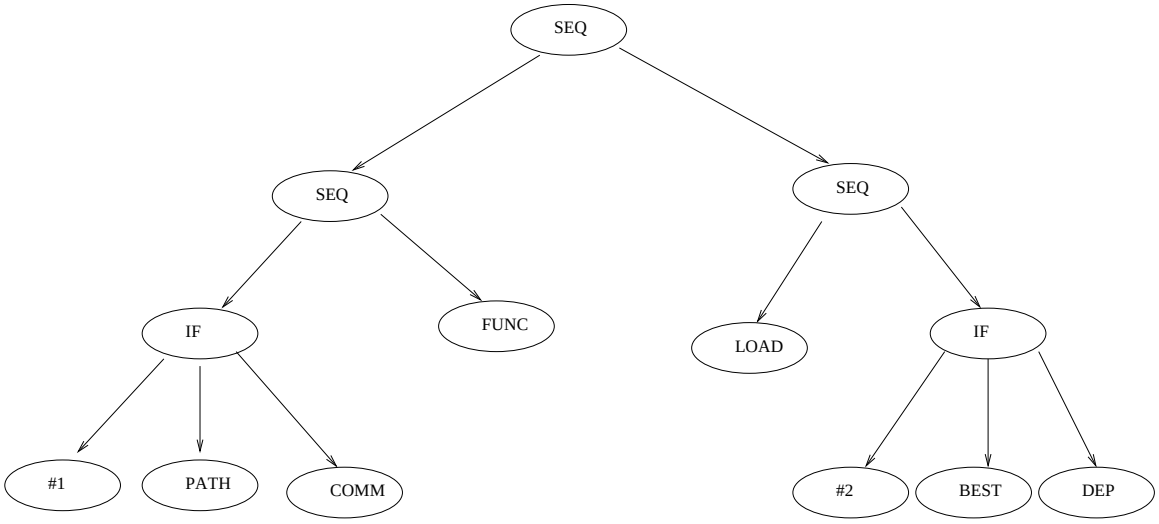


Figure 4-2: Sequence of passes coded as an s-expression. The s-exp represents a sequence that runs (PATH) or (COMM), then (FUNC) and (LOAD), and then (BEST) or (DEP), according to the results computed by the metrics.

Chapter 5

Compiler Infrastructure

In this chapter, we describe our compiler infrastructure and how we integrated convergent scheduling into Raw and Chorus. We also describe the scheduling algorithms we used for comparison in our experiments.

5.1 Chorus Infrastructure

The Chorus clustered VLIW system is a flexible compiler/simulator environment that can simulate a large variety of different configurations of clustered VLIW machines.

In chapter 6, we use it to simulate a clustered VLIW machine with four identical clusters. Each cluster has four functional 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.

Nonetheless, most of these parameters can be changed in our infrastructure. We tested the robustness of convergent scheduling to these changes in chapter 7.

As part of the Chorus infrastructure, we developed a flexible VLIW scheduler, seamlessly integrated with the rest of the system [Maz01]. The scheduler was written in C++, using the Machsuif infrastructure [Smi00].

The system (see figure 5-1):

- reads in the program, and build the dependence DAG for every block;
- determines the best space-time scheduling for the instructions in the block; different algorithms are used and compared;
- tags instructions with the chosen space/time schedule;
- sorts instructions so to be in the correct final sequence;
- creates new names for the temporary registers used in clusters different from the first and corrects the instructions using the new names;
- performs register allocation.

Register allocation is performed by MachSuif Register Allocator (based on George and Appel's work [GA96]), modified by our group to manage multiple clusters and predication. The code is then finalized and simulated on our step-by-step simulator.

In this work, three algorithms are implemented and tested. Convergent scheduling is the first. As a comparison, we implemented Desoli's PCC algorithm for clustered DSP architectures [Des98], and the UAS algorithm [OBC98].

5.1.1 Partial Component Clustering (PCC) algorithm implementation

Our system implements Desoli's Partial Component Clustering (PCC) algorithm, as in [Des98]. We try to illustrate our implementation here in detail.

Our algorithm builds the sub-components, as described in the paper. Then, it initially assigns them so to balance the load in different cluster. As in the original implementation, we iterate the initial assignment changing the size of ϕ_{th} (the maximum size of a sub-component) to minimize the expected schedule. This is done using a simplified list scheduler, more details about which are below. In the paper, the way that the various values of ϕ_{th} are chosen and the test to stop iterating are

Figure 5-1: Chorus VLIW Scheduler

not described. We choose a small set of different sizes that, in our tests, offered the best results: 1/80, 1/50, 1/20, 1/5 of the graph size, with a minimum of 5 nodes. Enlarging the set of ϕ_{th} would heavily slow down the process of compiling, already extremely slow with this algorithm (see section 6.5).

After the initial assignment, the partition is changed by moving one sub-component from one cluster to another. The algorithm performs a simple iterative descent. The first criterion of descent is the expected length of the schedule, the second criterion is the number of inter-cluster copies. As for the original paper, we perform a descent, and we stop when there is no further improvement. The described descent is a very computing-intensive process: for every trace, the algorithm computes the expected schedule length for every possible assignment to clusters.

After partitions are built, a simple list scheduler is run. Instructions are prioritized with the information about their criticality in the block and register pressure. The scheduler concurrently schedules and manages communication, by introducing the required transfer instructions.

The simplified list scheduler used in the iterative part does not explicitly creates the copy instructions, but models the delay of inter-cluster communication. The algorithm properly accounts for preplacement information. It does so by modeling the extra costs incurred by the clustered VLIW machine for a non-local memory access.

5.1.2 Unified Assign-and-Schedule

We implemented UAS as described in [OBC98]. UAS performs partitioning as part of the main cycle of its list scheduler (see figure 5-2).

In the outer loop of the list scheduler, instructions are prioritized with critical path information and register pressure. In the inner loop, different priority schemes are known for ordering the available clusters.

To take advantage of the information about preplacement, we slightly modified the CPSC heuristic described in [OBC98]. In addition to considering critical path and location of the parents, we prioritize the cluster where the instructions are preplaced.

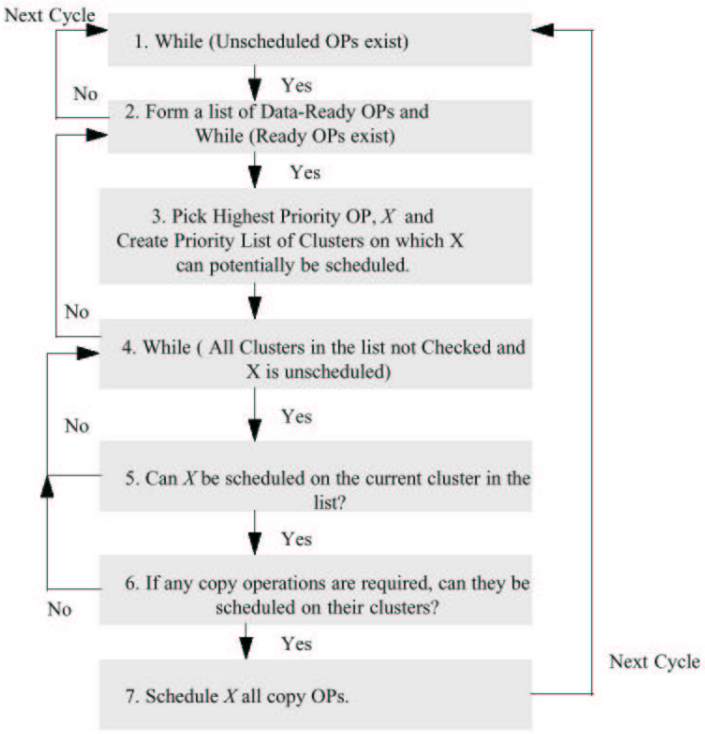


Figure 5-2: UAS algorithm (from [OBC98])

This reduces the average memory latency, because increases the number of memory operations issued to the cluster local memories.

5.2 RAW architecture

Figure 5-3 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 among the register file/ALUs on different tiles. 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

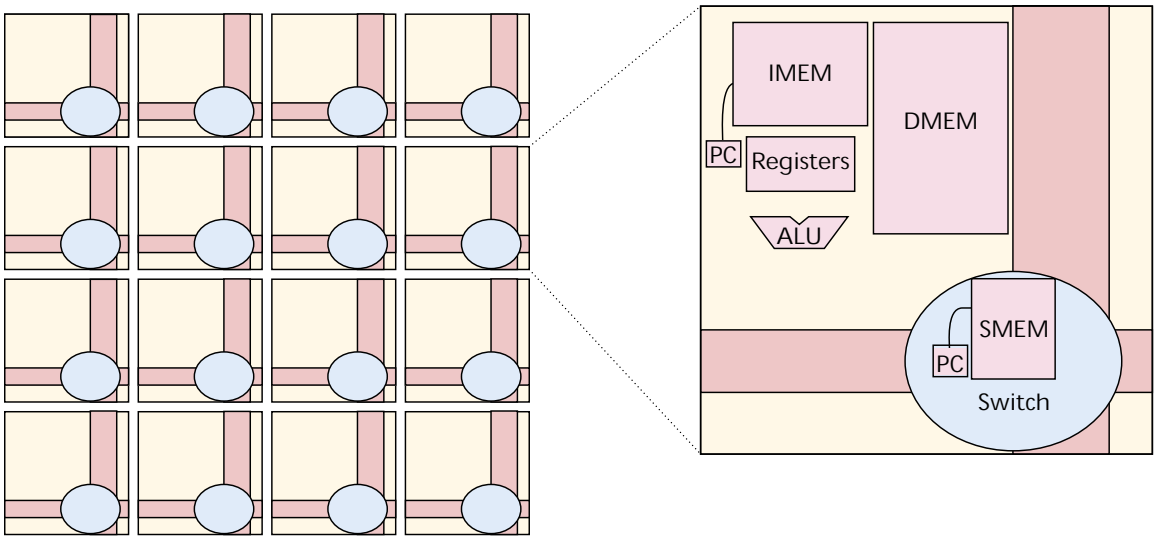


Figure 5-3: The Raw machine.

hop takes one extra cycle of latency.

Rawcc, the Raw compiler [LBF+98], takes a sequential C or Fortran program and parallelizes it across Raw tiles. It is built on top of the Machsuif intermediate representation [Smi00].

The compiling tool-chain has three main phases: the first phase is responsible for high level analysis and transformations, such as memory disambiguation, loop unrolling and array reshape; the second phase performs space-time scheduling; the third phase generates code for the processors and the switches.

The first phase identifies one or more scheduling traces, which are single-entry single-exit predicated blocks. For each trace, Rawcc constructs the data precedence graph and performs space-time scheduling on each graph. This process is called *orchestration* in the compiler, because every block is broken into a set of intercommunicating basic blocks, which the compiler has the task to coordinate in order to guarantee semantics, performance and absence of communication bottlenecks.

The space-time scheduling is split into an initial phase of spatial scheduling, followed by time scheduling. The spatial scheduling is further broken into partitioning and placement.

Partitioning consists in identifying parallel streams of computation within the

scheduling unit. The compiler tries to partition instructions in order to minimize final run-time. It assumes non-zero communication cost but infinite processing resources. The communication is modeled with uniform latency. Instruction clusters that communicate intensively among each other are then merged together if the grain of computation is too small to take advantage of the communication network. This process is based on the Dominant Sequent Clustering algorithm [GY94].

Then, the compiler merges the clusters, trying to minimize expected communication and total running time, until the number of clusters is equal to the number of tiles. After this, placement is done by mapping the instruction clusters to the different tiles using a greedy algorithm that minimized total communication. Finally, time scheduling is done by a greedy list scheduler, which orchestrates the necessary communication among tiles.

While very effective, we noticed two main problems with this approach:

- preplaced instructions (see next section) are not dealt effectively;
- partitioning does not consider the effects of time scheduling: phase ordering is a major problem in Rawcc.

The results shown in the next chapters were collected using btl, the cycle-accurate simulator of Raw developed at MIT. It can simulate the execution of our benchmarks on a Raw configuration that reproduces the .15 μm 16-tile prototype ASIC chip.

5.3 Preplacement Analysis

Both Rawcc and the Chorus compilers employ congruence transformation and analysis to increase the predictability of memory references [LA02]. 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.[1] On Chorus, all values that are live across multiple scheduling regions are mapped to the first cluster.

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

Chapter 6

Results: Comparison with State-of-the-art Scheduling Techniques

We have implemented convergent scheduling in two systems: the Raw architecture [TKM+02] and the MIT Chorus clustered VLIW infrastructure [Maz01].

6.1 Convergent schedulers

Table 6.1 lists the heuristics used by convergent scheduling 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.

Both Chorus and Rawcc use the spatial assignments given by the convergent scheduler to determine partitions. Chorus uses the temporal assignments as priorities

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

Table 6.1: Sequence of heuristics used by the convergent scheduler for (a) the Raw machine and (b) clustered VLIW.

for the list scheduler. For Rawcc, however, the temporal assignments are computed independently by its own instruction scheduler.

6.2 Benchmarks

Our sources of benchmarks include the Raw benchmark suite (jacobi, life) [BFL+97], 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 affect the results qualitatively. See table 6.2 for more details.

6.3 Performance comparisons

We compared our results with the baseline Rawcc and Chorus compilers. Table 6.3 compares the performance of convergent scheduling to Rawcc for two to 16 tiles. Figure 6-1 plots the same data for 16 tiles. Results show that convergent scheduling consistently outperforms baseline Rawcc for all tile configurations for most of the

Benchmark name	Suite	Number of lines	Exec. time (cycles)
Cholesky	Nasa7/Spec92	128	37 millions
FIR	RAW suite	59	5 millions
mxm	Nasa7/Spec92	67	170,000
rbsorf	RAW suite	22	273 millions
tomcatv	Spec95	193	67 millions
vvmul	RAW suite	19	29 millions
YUV	RAW suite	26	1.5 millions
vpenta	Nasa7/Spec92	195	6.5 millions
fpppp	Spec95	1029	150,000
SHA	RAW suite	625	1 million

Table 6.2: Characteristics of tested benchmarks. Execution time is computed on one-cluster configurations.

	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 6.3: Speedup on Raw. Speedup is relative to performance on one tile.

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 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

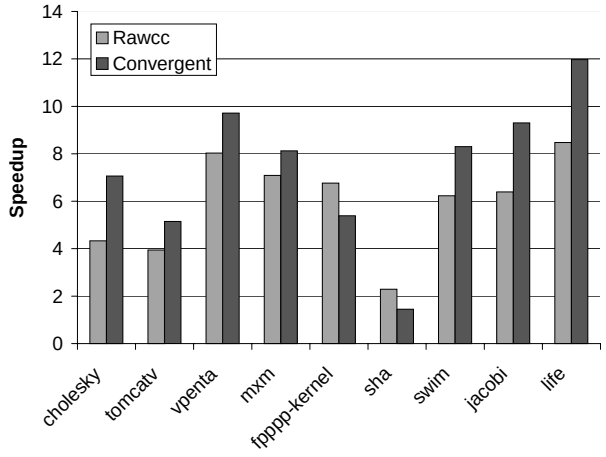


Figure 6-1: Performance comparisons between Rawcc and Convergent scheduling on a 16-tile Raw machine.

than the clustering counterpart in Rawcc — we expect that integrating a clustering pass to convergent scheduling will address this problem.

Figure 6-2 compares the performance of convergent scheduling to two existing assignment/scheduling techniques for clustered VLIW: UAS [OBC98] and PCC [Des98]. We augmented each existing algorithm with preplacement information. For UAS, we modified 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.

6.4 Scheduling convergence

We were interested in determining the effectiveness and the importance of different passes in modifying and affecting the final schedules. To do this, we measured how many instructions were moved from one cluster to another by each pass. With this,

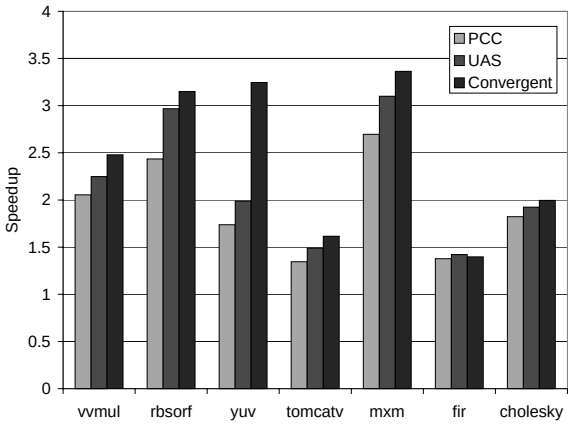


Figure 6-2: Performance comparisons between PCC, UAS, and Convergent scheduling on a four-clustered VLIW. Speedup is relative to a single-cluster machine.

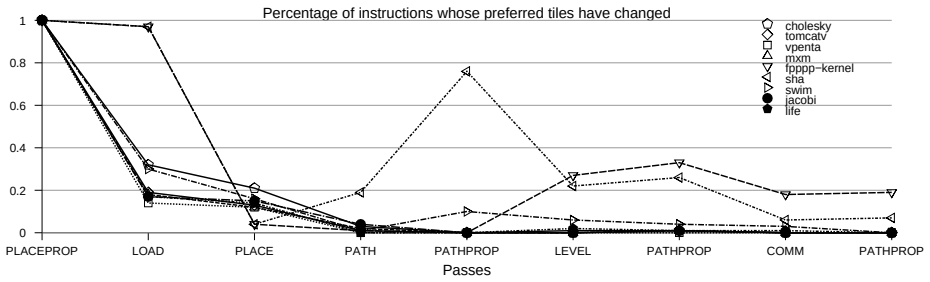


Figure 6-3: Convergence of spatial assignments on Raw.

we mean the number of instructions for which the preferred cluster before and after a specific pass was different.

Figure 6-3 shows the percentage of instructions whose preferred tiles are changed by each convergent pass on Raw. The plot measures static instruction counts, and excludes 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 6-4 shows the percentage of static instructions whose preferred tiles are

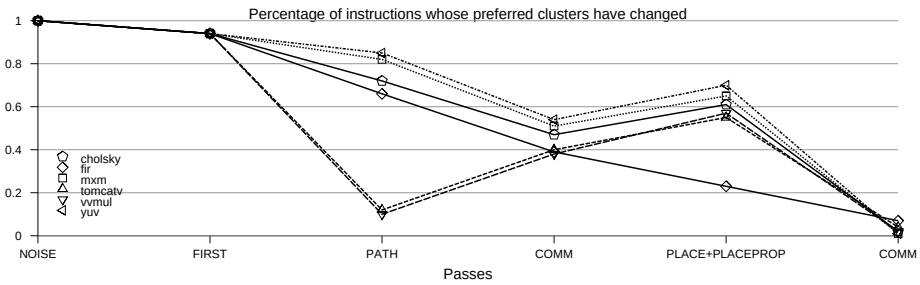


Figure 6-4: Convergence of spatial assignments on Chorus.

changed by each convergent pass on Chorus. Again, passes that only modify temporal preferences are excluded. Here we note that PATH does not affect heavily tomcatv and vvmul, which feature very high parallelism when unrolled. PLACE is important for most benchmarks, except FIR, for which our preplacement analysis is not very effective.

6.5 Compile-time scalability

We examined 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 6-5 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

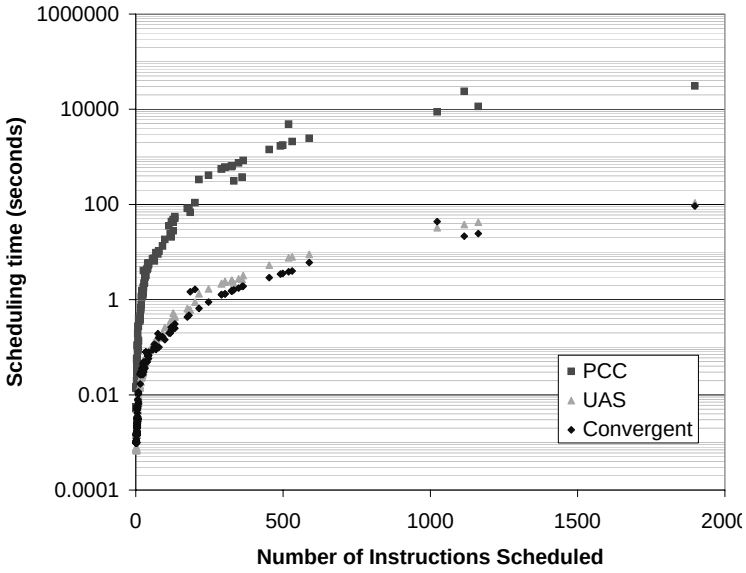


Figure 6-5: Comparison of compile-time vs input size for algorithms on Chorus.

results in much poorer assignment quality.

Chapter 7

Results: Adapting to Different Architectures

This chapter describes our compilation framework as well as the methodology we use to collect results. We begin by describing the GP parameters we use to train the convergent scheduler. We then describe the architectures we tested, and the results we obtained. We discuss the validity of our findings using cross-validation, and then we give a summary of results.

7.1 GP Parameters

We wrapped the GP framework depicted in Figure 4-1 around our compiler and simulator. For each individual in the population, our *harness* compiles the benchmarks in our training suite with the phase ordering described by its *genome*. The specific parameters we use are summarized in Table 7.1. All experiments maintain a population of 200 individuals. After every generation we discard the weakest 20% of the population, and replace it with new individuals. Of these new genomes, half of them are complelety random, and the remainder are created via the crossover operator. 5% of the individuals created via crossover are subjected to mutation. Finally, we run each experiment for 40 generations.

Fitness is measured as the average speed-up (over all the benchmarks in our

PARAMETER	VALUE
Mutated individuals	5%
Crossover v. random individuals	50%
Mortality	20%
Pop. size	200
Generations	40 / 100

Table 7.1: Parameters of the evolutionary framework

training suite) when compared against the phase ordering that we used in chapter 6. We also reward *parsimony* by giving preference to the shorter of two otherwise equivalently fit sequences.

7.2 Tested Architectures

We tested our system using a set of different architectures. As mentioned earlier, on Chorus we can easily change a number of machine characteristics with ease, including number of clusters, number and type of functional units, size of the register file, cost of inter-cluster communication and memory access time. We tested the robustness of convergent scheduler under these changes, in particular, we tested the following architectures:

- four clusters with limited communication;
- two clusters with limited communication;
- four clusters with limited registers.

7.2.1 Baseline (4cl)

The baseline architecture is a 4-cluster VLIW with rich interconnectivity. In this configuration, the clusters are fully connected with a 4x4 crossbar. Thus, the clusters can exchange up to four words every cycle. The delay for the communication is 1 cycle. Register file, functional units and L1 cache are split into the clusters – even if every address of the memory can be accessed by any cluster – with a penalty of

1 cycle for non-local addresses. The cache takes 6 cycles to access and the register file takes 2 cycles. In addition, memory writes take 1 cycle. Each cluster has 64 general-purpose registers and 64 floating-point registers.

7.2.2 Limited bus (4cl-comm)

This architecture is similar to the baseline architecture, the only difference being inter-cluster communication capabilities. This architecture only routes one word of data per cycle on a shared bus, which can be snooped, creating so a basic broadcasting capability. Because this model has limited bandwidth, the space-time scheduler must be more conservative in splitting computation across clusters.

7.2.3 Limited bus (2cl-comm)

Another experiment uses an architecture that is substantially weaker than the baseline. It is the same as machine 4cl-comm, except it only has 2 clusters.

7.2.4 Limited Registers (4cl-regs)

The final machine configuration on which we test our system is identical to the baseline architecture, except that each cluster has half the number of registers (32 general-purpose and 32 floating-point registers).

The next section discusses experimental results using the machine configurations described above.

7.3 Results

As in the previous chapter, we compared the performance of convergent scheduling with UAS and PCC. In all of our experiments, `(inittime)` is hardwired to be the first pass, as part of the initialization, and `(place)` is always run at the end of the sequence, to guarantee semantics.

7.3.1 Baseline (4cl)

The baseline sequence was hand-tuned in our initial work with convergent scheduling. As seen in the previous chapter, this sequence outperforms UAS and PCC in the baseline configuration.

```
(inittime) (noise) (first) (path) (comm) (place) (placeprop) (comm)
(emphcp)
```

7.3.2 Limited Bus (4cl-comm)

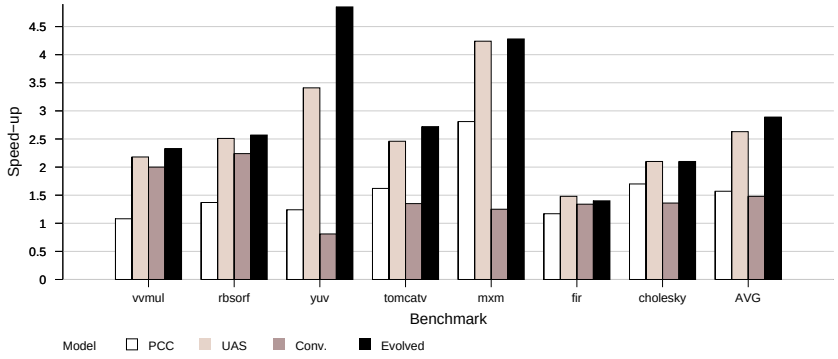
We use this configuration to perform many experiments. We evolved a sequence for 100 generations, with 200 individuals, over seven representative benchmarks.

Figure 7-2 plots the fitness of the best creature over time. The fitness is measured as the average (across benchmarks) normalized completion time with respect to the sequence for our baseline architecture. The sequence improves quickly in the first 36 generations. After that, only minor and slow improvements in fitness could be observed. This is why, in our cross-validation tests (see section 7.4), we limit our evolution to 40 generations.

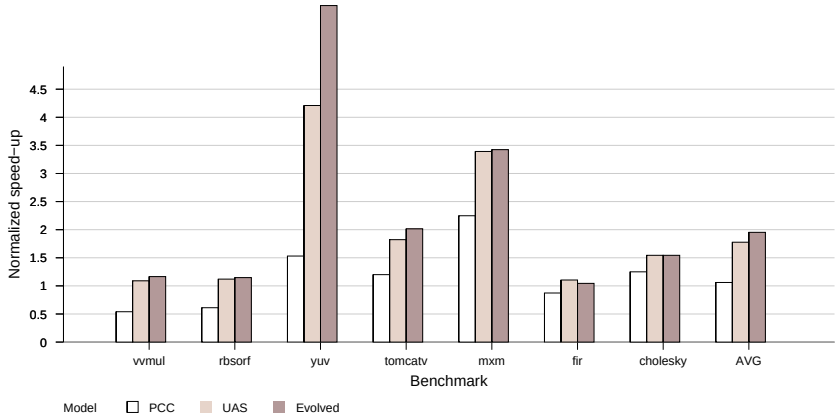
The evolved sequence is more conservative in communication and stresses the importance of the `(comm)` pass – it is run three times. Also, `(dep)` and `(func)` are important: `(dep)`, as a side effect, increases the probability that two dependent instructions are scheduled next to each other in space and time; `(func)` reduces peaks on overloaded clusters, which could lead to high amounts of localized communication.

```
(inittime) (func) (dep) (func) (load) (func) (dep) (func) (comm)
(dep) (func) (comm) (place)
```

The plot in figure 7-1 compares the evolved sequence with the original sequence and our reference schedulers. The evolved sequence performs about 10% better than UAS, and about 95% better than the sequence tuned for the baseline architecture. In this test, PCC performed extremely poorly, probably due to limitations in the



(a)



(b)

Figure 7-1: (a)Speedup on 4cl-comm compared with 1-cluster convergent scheduling (original sequence). In the graph, conv. is the baseline sequence, evolved is the new sequence for this architecture. (b) Performance normalized to the baseline sequence.

modeling of communication done by our implementation of the internal simplified scheduler (see [Des98]).

7.3.3 Limited bus (2cl-comm)

`(inittime) (dep) (noise) (func) (noise) (noise) (comm) (func) (dep) (func) (place)`

Similar to the previous tests, `(comm)`, `(dep)` and `(func)` are important in creating a smooth schedule. We notice the strong presence of `(noise)` in the middle of the sequence. It appears as if the pass is intended to move away from local minima by

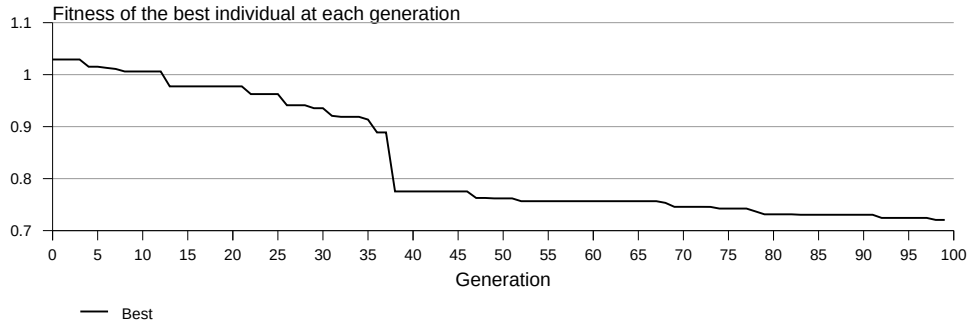


Figure 7-2: Fitness of the best individual, during evolution on 4cl-comm.

shaking up the schedule.

The evolved sequence stands the comparison with UAS (about 4% better) and PCC (about 5% better). Here PCC does not show the same problems present with 4cl-comm (see figure 7-3). We observe an improvement of 12% over the baseline sequence.

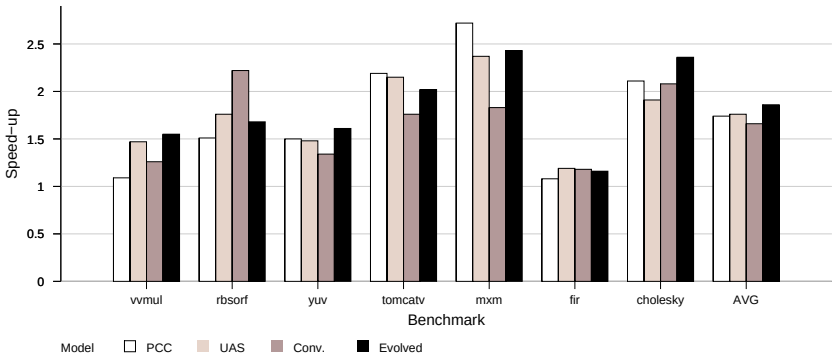
7.3.4 Limited Registers (4cl-regs)

Figure 7-4 shows the performance of the evolved sequence when compared with our baseline and our reference. We can measure an improvement of 68% over the baseline sequence. Here again, `(func)` is a very important pass. UAS outruns convergent scheduling in this architecture by 6%, and PCC by 2%.

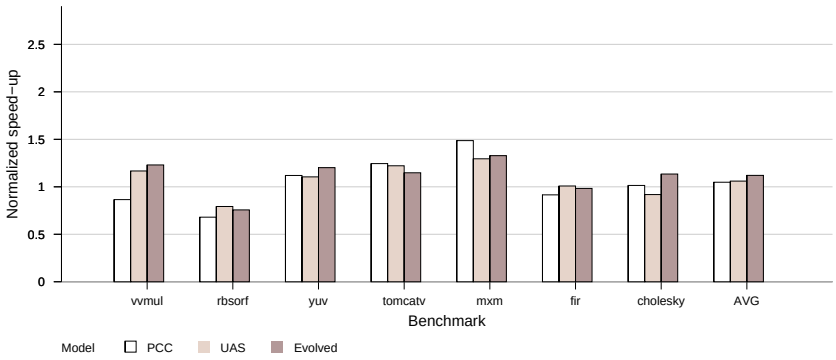
```
(inittime) (func) (dep) (func) (func) (func) (func) (path) (func)
(place)
```

7.4 Leave-one-out Cross Validation

We tested the robustness of our system by using leave-one-out cross validation on 4cl-comm. In essence, cross validation helps us quantify how applicable the sequences are when applied to benchmarks that were not in the training set. The evolution was rerun excluding one of the seven benchmarks, and the result tested again on the excluded

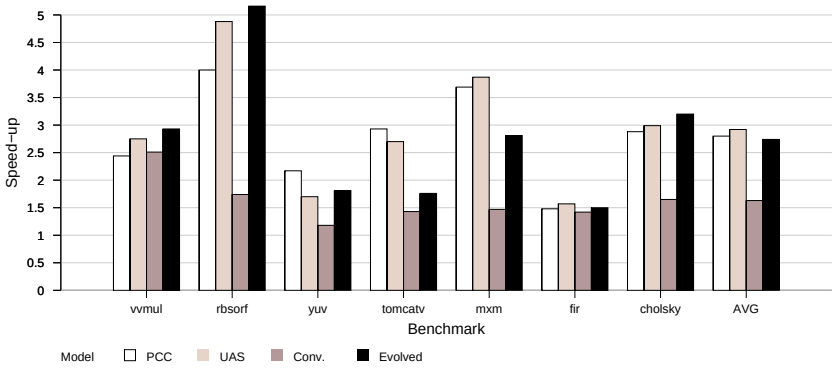


(a)

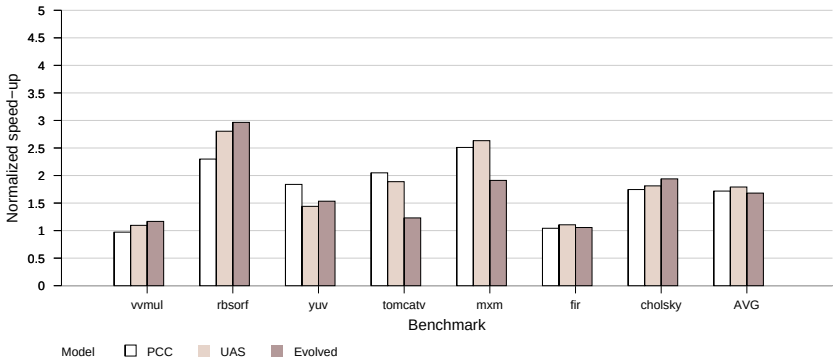


(b)

Figure 7-3: Speedup on 2cl-comm.



(a)



(b)

Figure 7-4: Speedup on 4cl-regs.

benchmark	Excluded benchmark cholesky	fir	yuv	tomcatv	mxm	vvmul	rbsorf	full
cholesky	**2.18**	2.18	2.18	2.18	2.18	2.17	2.18	2.18
fir	1.35	**1.35**	1.35	1.35	1.35	1.35	1.35	1.35
yuv	1.53	1.53	**1.53**	1.53	1.53	1.16	1.53	1.53
tomcatv	1.60	1.35	1.35	**1.45**	1.47	1.55	1.44	1.37
mxm	2.03	2.04	2.04	2.04	**2.12**	2.33	2.04	1.96
vvmul	2.18	2.18	2.18	2.18	2.18	**2.25**	2.18	2.18
rbsorf	2.41	2.41	2.41	2.44	2.36	2.44	**2.41**	2.41
average	1.90	1.86	1.86	1.88	1.89	1.89	1.88	1.86

Table 7.2: Results of cross validation. The highlighted numbers refer to the benchmark excluded in each test.

Excluded benchmark	Sequence
cholesky	`(inittime) (comm) (load) (comm) (load) (func) (place)`
fir	`(inittime) (func)(place)`
yuv	`(inittime) (func)(place)`
tomcatv	`(inittime) (func) (best) (place)`
mxm	`(inittime) (best) (best) (best) (func) (place) (place)`
vvmul	`(inittime) (func) (dep) (func) (place)`
rbsorf	`(inittime) (best) (func) (place)`
none excluded (40 gens.)	`(inittime) (comm) (best) (func) (place)`

Table 7.3: The sequence evolved in our cross-validation tests.

benchmark. In table 7.2, the results are shown. The seven cross-validation evolutions reached results very similar to the full evolution, for the excluded benchmarks too (within 6% of the performance). The seven evolved sequences (in table 7.3) are all similar: `(func)` is the most important pass for this architecture.

7.5 Summary of Results

We verified that convergent scheduling is well suited to a set of different architectures. Running on 20 dual Pentium 4 machines, evolution takes a couple of days.

Sequences that contain conditional expressions never appeared in the best individuals. It turns out that running a pass is more beneficial than running a test to condition its execution. This is largely because convergent scheduling passes are somewhat symbiotic by design.

In other words, the results show that passes do not disrupt good schedules. So, running extra passes is usually not detrimental to the final result. We verified that running a complex measurement can take as much time as running a simple pass. Therefore, when measuring the complexity of resulting sequences, we assign equal weight to passes and tests. Our bias for shorter genomes (parsimony pressure) penalizes sequences with extra tests as well as sequences with useless passes. In the end, conditional tests were not used in the best sequences. Rather, all passes are unconditionally run.

Nevertheless, we still believe in the potential of this approach, and leave further exploration to future work.

Chapter 8

Related work

The problem of phase ordering, and in particular the interaction between register allocation and scheduling, have been discussed by many authors. Lerner et al. propose an interesting interface to different passes based on *graph replacement* [LGC02]. Their approach enables passes to be designed independently, but to run together. The authors claim to be able to reach a precision close to manually built super-passes. In [MPSR95], the authors introduce a combined register allocation and instruction scheduling heuristic that is shown to be easier to approximate and performs 16-21% better than the two independent optimizations. The approach simply weights the result of the two heuristics to obtain a single value, which is used to choose instructions. Register allocation and scheduling are combined also in [JC98], where the authors introduce an algorithm that performs the two operations together, with encouraging results.

Many heuristics approaches have been developed for space-time scheduling. UAS (Unified Assign-and-Schedule) performs space-time scheduling on clustered VLIWs in a single step, using a greedy, list-scheduling-like algorithm [OBC98]. Desoli describes an algorithm targeted for graphs with a large degree of symmetry [Des98]. Leupers describe an iterative combined approach to perform scheduling and partitioning on a VLIW DSP [Leu00]. The approach is based on simulated annealing.

Integrated approaches to partitioning, scheduling and register allocation can also be found in the area of modulo scheduling for clustered VLIW architectures [SG00,

ZLAV01].

However, there have been far fewer space-time scheduling algorithms that take into account preplaced instructions. One such algorithm is BUG (Bottom-Up-Greedy). BUG is implemented on ELI, one of the earliest spatial architectures that relies on the compiler for space-time scheduling [Ell85]. BUG only performs space-scheduling; time-scheduling is done via traditional list scheduling. BUG is a two-phase algorithm: the algorithm first traverses a dependence graph bottom-up to propagate information about preplaced memory instructions. Then, it traverses the graph top-down and greedily map each instruction to the clusters that can execute it earliest. The Multiflow compiler uses a variant of BUG [LFK+93], but it does not account for preplaced instructions. Lee also handles preplaced instructions [LBF+98]. He borrows his general approach from multiprocessor task graph scheduling [LBF+98]. Like Ellis, Lee uses a separate list scheduler to perform time-scheduling. Space-scheduling 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. Constraints from preplaced instructions are mainly handled during placement.

Many researchers have used machine-learning techniques to solve hard compilation problems. Therefore, only the most relevant works are discussed here. Cooper et al. use a genetic-algorithm solution to evolve the order of passes in an experimental compiler [CSS99]. Our research extends theirs in many significant ways. First, our learning representation allows for conditional execution of passes, while theirs does not. In addition, we differ in the end goal; because they were targeting embedded microprocessors, they based fitness on code size. While this is a legitimate metric, code size is not a big issue for parallel architectures, nor does it necessarily correlate with wall clock performance. We also simultaneously train on multiple benchmarks to create general-purpose solutions. They train on one benchmark at a time, and thus create application-specific phase orderings. Finally, we believe the convergent scheduling solution space is more interesting than that of an ordinary backend. The symmetry and unselfishness of convergent scheduling phases implies an interesting

and immense solution space.

Calder et. al used supervised learning techniques to fine-tune static branch prediction heuristics [CaMJL+97]. They employ two learning techniques — neural networks and decision trees — to search for effective static branch prediction heuristics. While our methodology is similar, our work differs in several important ways. Most importantly, we use unsupervised learning, while they use supervised learning. Unsupervised learning is used to capture inherent organization in data, and thus, only input data is required for training. Supervised learning learns to match training inputs with known outcomes. This means that their learning techniques rely on knowing the optimal outcome, while ours does not. Our problem demands an unsupervised method since optimal compiler sequences are not known.

The COGEN(t) compiler creatively uses genetic algorithms to map code to irregular DSPs [GW01]. This compiler, though interesting, evolves on a per-application basis. Nonetheless, the compile-once nature of DSP applications may warrant the long, iterative compilation process.

Chapter 9

Future Work and Conclusion

Time-to-market pressures make it difficult to effectively target next generation processors. Convergent scheduling's simple interface alleviates such constraints by offering a set of innovative features:

- a simple interface that allows rapid prototyping of new passes,
- the opportunity for passes to express a preference for a schedule, rather than a simple black-or-white decision,
- the opportunity for passes to easily override and reverse previous decisions, if needed,
- an interface that allows an abundant exchange of information between passes,
- a framework that allows passes to be run in different orders without effort,
- the opportunity to let the compiler adapt to a different architecture with ease.

With this infrastructure, we were able to improve speed-up for two spatial architectures: Raw, 21% better, and the MIT Chorus clustered VLIW architecture, 14% better than UAS and 28% better than PCC in our baseline model.

Because the scheduler's framework allows passes to be run in any order, there are countless legal phase orders to consider. This thesis showed how machine-learning

techniques could be used to automatically search the phase-order solution space. Our genetic programming technique allowed us to easily re-target new architectures.

In our work, we also experimented with learning dynamic policies. Instead of choosing a fixed static sequence of passes, our system is capable of dynamically choosing the best passes for each scheduling unit, based on the status of the schedule. Although the learning algorithm did not find sequences that conditionally executed passes, we still have reason to believe that this is a promising approach. Future work will explore this in greater detail.

Our technique was able to find architecture-specific phase orders which improved execution time by 12% to 95%. Cross validation showed that performance improvement is not limited to the benchmarks on which the sequence was trained.

As part of the future work, we would like to include register allocation in our preference maps, and let convergent scheduling perform this task too.

Also, we are interested in an approach based on reinforcement learning. Based on the current status of the prefence maps, the compiler should learn which pass to run next. After learning, the compiler will consist of a policy, able to pick up different passes as needed, till the schedule converges.

This way, instead of running a predefined sequences of passes (with or without conditionals), the system will choose dynamically the best pass for the given block. We believe this could be a major breakthrough. We also consider how to augment the currently used compilers to collect information and to continuously learn about the compiling process itself.

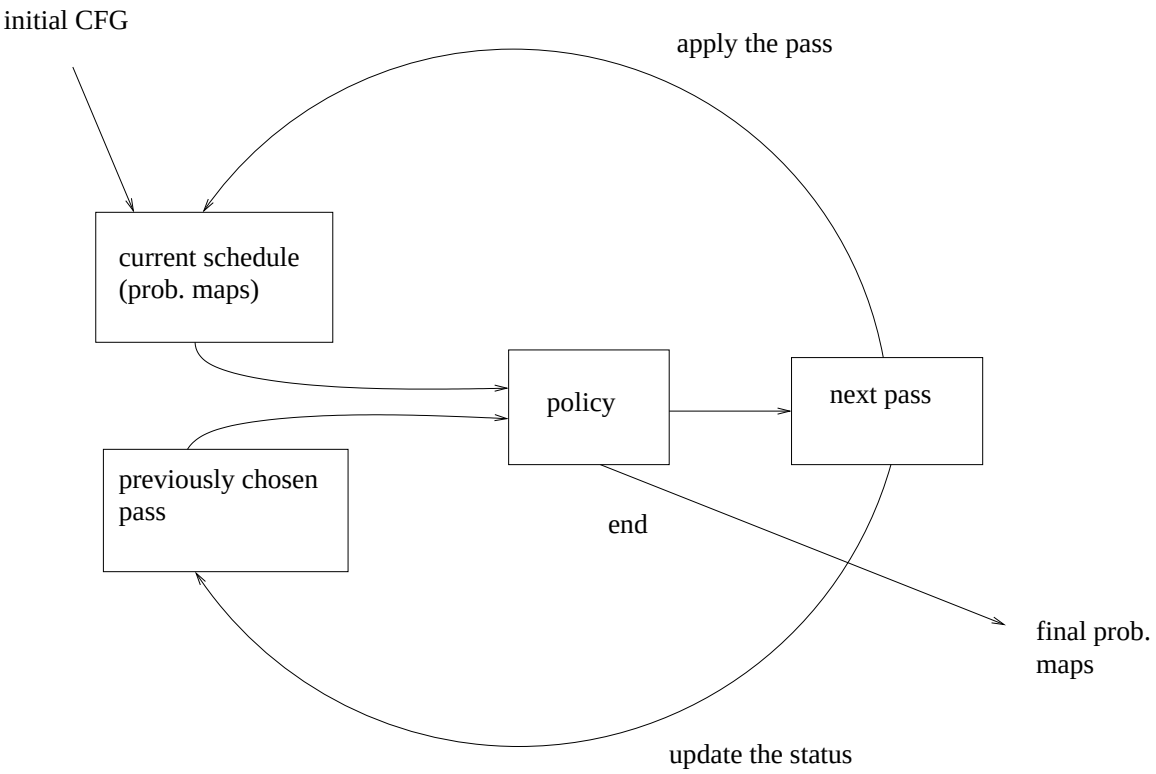


Figure 9-1: A compiler with dynamic policy for choosing passes

Bibliography

[BFL+97] Jonathan Babb, Matthew Frank, Victor Lee, Elliot Waingold, Rajeev Barua, Michael Taylor, Jang Kim, Srikrishna Devabhaktuni, and Anant Agarwal. The raw benchmark suite: Computation structures for general purpose computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, April 1997.

[BLAA99] Rajeev Barua, Walter Lee, Saman Amarasinghe, and Anant Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[CaMJL+97] Brad Calder, Dirk Grunwald ad Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-Based Static Branch Prediction Using Machine Learning. In *ACM Transactions on Programming Languages and Systems (ToPLaS-19)*, volume 19, 1997.

[CSS99] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *ACM Proceedings of the SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 1999.

[Des98] Giuseppe Desoli. Instruction assignment for clustered vliw dsp compilers: a new approach. Technical Report HPL-98-13, Hewlett Packard Laboratories, January 1998.

[Ell85] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Yale University, 1985.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. In *ACM Transactions on Programming Languages and Systems*, volume 18, 1996.

[GW01] Gary William Grewal and Charles Thomas Wilson. Mappping Reference Code to Irregular DSPs with the Retargetable, Optimizing Compiler COGEN(T). In *International Symposium on Microarchitecture*, volume 34, pages 192–202, 2001.

[GY94] A. Gerasoulis and Tao Yang. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transaction on Parallel and Distributed Systems*, 5:951–967, 1994.

[JC98] J. Janssen and H. Corporaal. Registers on demand: an integrated region scheduler and register allocator. In *Conference on Compiler Construction*, 1998.

[KH99] Matthew Kessler and Thomas Haynes. Depth-Fair Crossover in Genetic Programming. In *Proceedings of the ACM Symposium on Applied Computing*, pages 319–323, February 1999.

[Koz92] John Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, 1992.

[KS02] Ho-Seop Kim and James E. Smith. An Instruction Set and Microarchitecture for Instruction Level Distributed Processing. In *Proceedings of the 29th International Symposium on Computer Architecture*, Anchorage, AL, May 2002.

[LA02] Sam Larsen and Saman Amarasinghe. Increasing and detecting memory address congruence. In *Proceedings of 11th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Charlottesville, VA, September 2002.

[LBF+98] Walter Lee, Rajeev Barua, Matthew Frank, Devabhatuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, San Jose, CA, October 1998.

[Leu00] Rainer Leupers. Instruction scheduling for clustered vliw dsps. In *International Conference on Parallel Architecture and Compilation Techniques, Philadelphia, PA, Oct. 2000.*, 2000.

[LFK+93] 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, January 1993.

[LGC02] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, Portland, Oregon, 2002.

[Maz01] David Maze. Compilation infrastructure for vliw machines. Master's thesis, Massachusetts Institute of Technology, September 2001.

[MPSR95] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining register allocation and instruction scheduling. Technical Report CS-TN-95-22, 1995.

[NSBK01] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler. A design space evaluation of grid processor architectures, 2001.

[OBC98] Emre Ozer, Sanjeev Banerjia, and Thomas M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *International Symposium on Microarchitecture*, pages 308–315, 1998.

[SAMO02] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta-optimization: Improving compiler heuristics with machine learning. Technical Report MIT-LCS-TM-634, 2002.

[SG00] J. Sanchez and A. Gonzalez. Instruction scheduling for clustered vliw architectures. In *13th International Symposium on System Sythesis (ISSS)*, Madrid, Spain, September 2000.

[Smi00] Micheal D. Smith. Machine suif. In *National Compiler Infrastructure Tutorial at PLDI 2000*, June 2000. http://www.eecs.harvard.edu/hube.

[TKM+02] Michael Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Jae-Wook Lee, Paul Johnson, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, pages 25–35, March/April 2002.

[WTS+97] Elliot Waingold, Micheal Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal. Baring it all to software: Raw machines. *Computer*, pages 86–93, September 1997.

[ZLAV01] Javier Zalamea, Josep Llosa, Eduard Ayguade, and Mateo Valero. Modulo scheduling with integrated register spilling for clustered vliw architectures. In *MICRO34*, 2001.