Journal of Instruction-Level Parallelism 1 (2004) 1-23 Submitted 6/04; published 9/04

Convergent Scheduling

Diego Puppin DIEGO.PUPPIN@ALUM.MIT.EDU
Institute for Information Science and Technologies
via Moruzzi 1,
56100, Pisa, Italy

Mark Stephenson MSTEPHEN@MIT.EDU
Walter Lee WALT@MIT.EDU
Saman Amarasinghe SAMAN@CAG.LCS.MIT.EDU
Laboratory for Computer Science, MIT
Technology Square,
Cambridge, MA, 02143 USA

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 passes, each implementing a heuristic that addresses a particular problem or constraint. The passes share a simple, common interface that allows the spatial and temporal preferences associated with each instruction to be queried and modified. 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 a hand-selected, fixed ordering of the passes we are able to obtain an average increase in speedup on a reference 4-cluster VLIW architecture of 28% when compared to Desoli's PCC algorithm, 14% when compared to UAS, and a speedup of 21% over the existing space-time scheduler of the Raw processor.

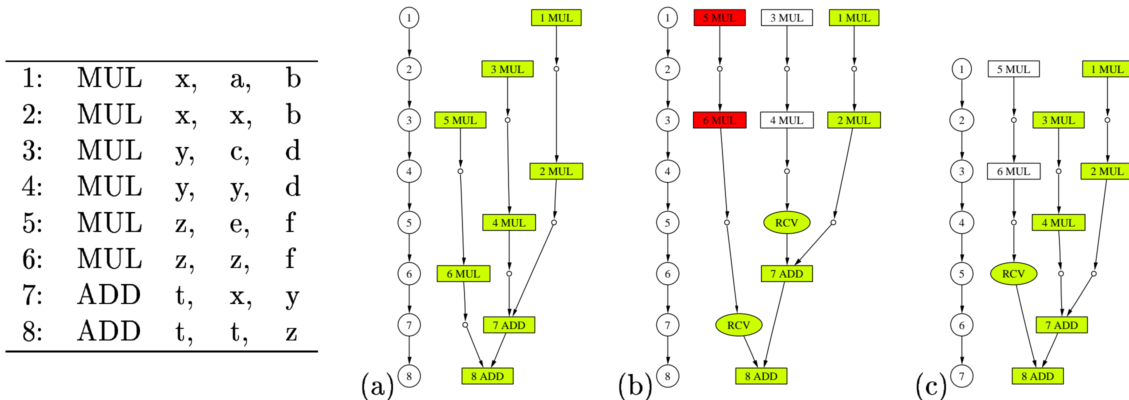
Then, we applied machine-learning techniques to automatically search for good pass orderings, when moving to different VLIW architectures. The architecture-specific pass orderings yield speedups ranging from 12% to 95% over the baseline order. The *cross validation* studies we ran show that our automatically 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.

1. Introduction

Instruction scheduling on modern microprocessors is becoming increasingly difficult. In almost all practical instances, it is NP-complete, and it often faces multiple contradictory constraints, e.g. code sequences that expose more instruction level parallelism (ILP) also have longer live ranges which induce higher register pressure.

To complicate matters, spatial architectures, such as clustered VLIWs, Raw [1], Grid processors [2], and ILDPs [3], distribute their computing resources. The extra degree of

©2004 AI Access Foundation and Morgan Kaufmann Publishers. All rights reserved.



1:	MUL	x,	a,	b
2:	MUL	x,	x,	b
3:	MUL	y,	c,	d
4:	MUL	y,	y,	d
5:	MUL	z,	e,	f
6:	MUL	z,	z,	f
7:	ADD	t,	x,	y
8:	ADD	t,	t,	z

Figure 1: An example of trade-off between parallelism and locality on spatial architectures. The code on the left is to be scheduled on a clustered architecture. Each cluster (represented by a different color) has one functional unit, and 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 trade-off between locality and parallelism.

scheduling freedom that such architectures introduce complicates the scheduling problem: communication costs between distant resources, communication resource contention, and increased register pressure must be taken into account. As shown in Figure 1, an effective scheduler must find the proper balance between parallelism and locality.

In addition, some instructions on spatial architectures have specific spatial requirements. These requirements arise from two sources. First, some load and store instructions must access memory banks on specific clusters, either for correctness or for performance reasons [4, 5]. Second, when a value is live across scheduling regions, its definitions and uses must be mapped to a consistent cluster [6]. We call instructions with these spatial requirements *preplaced instructions*. A good scheduler must be sensitive to constraints imposed by preplaced instructions in order to generate a good schedule.

A scheduler also faces difficulties because different heuristics work well for different types of graphs. Figure 2 depicts two representative data dependence graphs: 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. For long, narrow graphs, critical-path based heuristics are likely to work well. For fat, parallel graphs, it is more important to minimize communication and exploit the coarse-grain parallelism. To perform well for arbitrary graphs, a scheduler must have 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, 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 passes. This

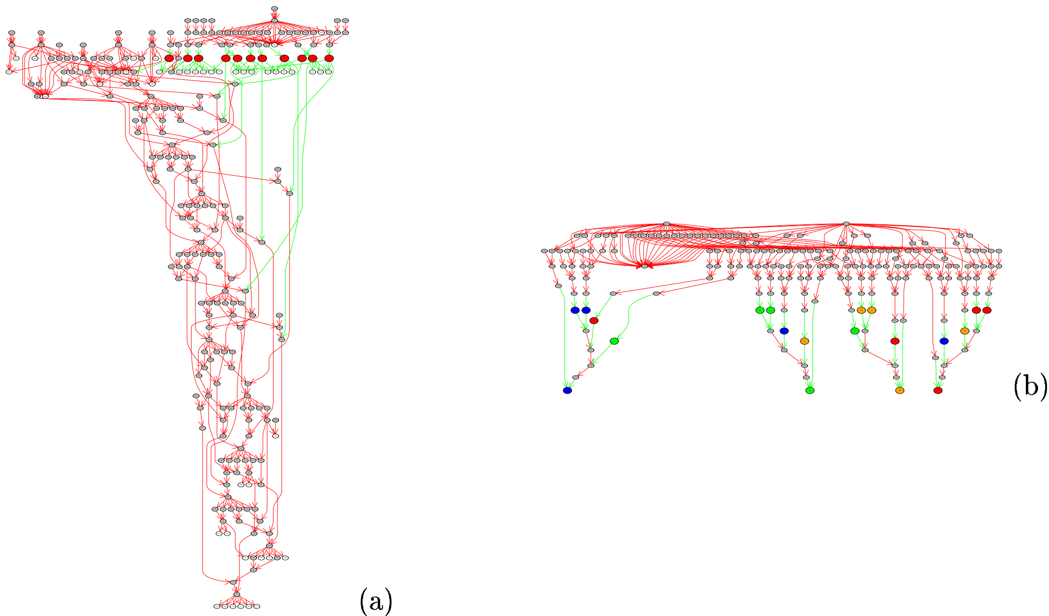


Figure 2: Example of data dependence graphs. Nodes represent instructions and edges represent data dependences between instructions.

approach, however, introduces pass ordering problems, as decisions made by the early passes are based on partial information and can adversely affect the quality of decisions made by subsequent passes. 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 [7]. However, extending such frameworks to support preplaced instructions is difficult – no such extension exists today.

This paper presents *convergent scheduling*, a radical departure from traditional scheduling methods. Convergent scheduling is a general scheduling framework that makes it easy to specify arbitrary constraints and scheduling heuristics. A convergent scheduler is composed of independent passes. Each pass implements a heuristic that addresses a particular problem such as ILP or register pressure.

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

Passes can be run multiple times, and in *any* order. Thus, while mitigating ordering problems due to hard constraints, a convergent scheduler is presented with a limitless number of legal pass orders. Initially, we tediously hand-tuned the pass order. Then, we applied machine learning techniques to automatically find good orderings for a convergent scheduler. Because different parallel architectures have unique scheduling needs, the speedups our system is able to obtain by creating architecture-specific pass orderings is impressive.

Equally impressive is the ease with which it finds effective sequences. In less than two days, using a modestly sized cluster of workstations, our system is able to discover architecture-specific sequences that produce speedups ranging from 12% to 95% over our hand-tuned sequence, and generally outperforms UAS [8] and PCC [9].

The contributions of this paper are:

- a novel interface between scheduling passes based on weighted preferences,
- a novel approach to address the combined problems of cluster assignment, scheduling, and register pressure,
- the formulation of a set of powerful heuristics to address both general constraints and architecture-specific issues,
- a demonstration of the effectiveness of convergent scheduling compared to traditional techniques,
- the use of machine learning to adapt convergent scheduling to new architectures in an effective way.

The rest of this work is organized as follows. Section 2 introduces convergent scheduling and uses an example to illustrates how it works. Section 3 describes the convergent scheduling interface between passes. Section 4 describes the collection of passes currently implemented in our framework. Then, Section 5 describes Genetic Programming, the machine-learning technique we use to explore the pass-order solution space. Sections 6 and 7 present our experimental results, using basic convergent scheduling, and using evolution with Genetic Programming. Section 8 gives an overview of related work. Finally, Section 9 concludes, summarizing the main achievements of our work.

2. Convergent Scheduling

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

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

Figure 3 shows how convergent scheduling operates on a small code sequence from fpppp. Initially, the weights are evenly distributed, as shown in (b). We apply the *noise*

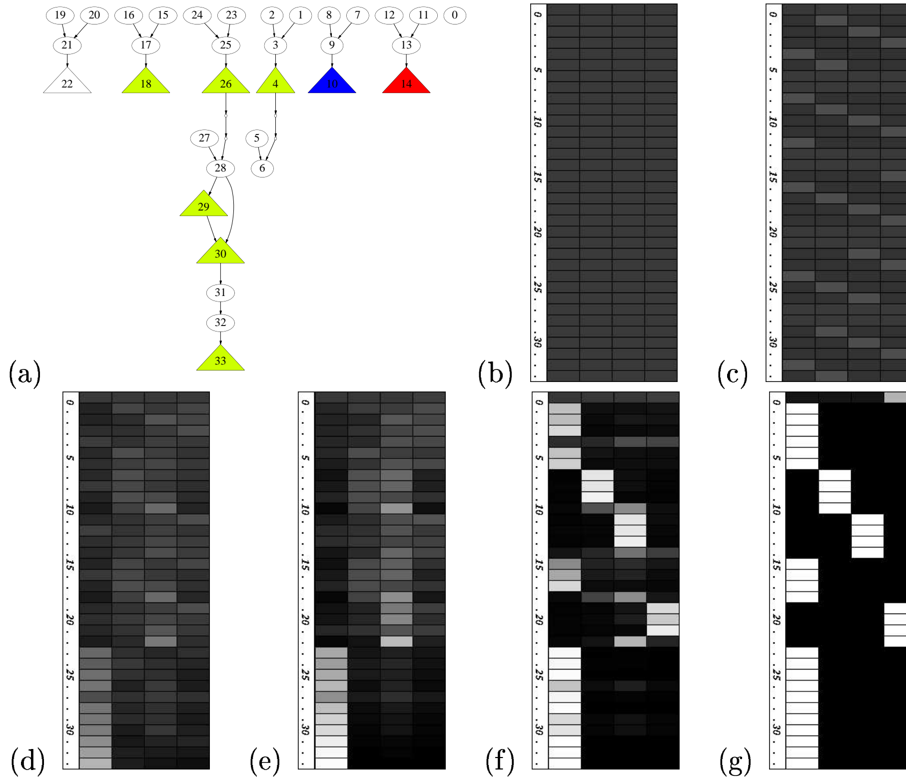


Figure 3: Convergent scheduling operates on a code sequence from fpppp. Figure (a) shows the data dependence graph representation of the scheduled code. Nodes represent instructions; edges represent dependences between instructions. Triangular nodes are preplaced, with different shades corresponding to different clusters. Figures (b)-(g) show how the convergent schedule is modified by a series of passes. This example only illustrates space scheduling, not time scheduling. Each figure in Figures 3b-g is a cluster preference map. A row represents an instruction. The row numbers correspond to the instruction numbers in (a). A column represents a cluster. The color of each entry represents the level of preference an instruction has for that cluster. The lighter the color, the stronger the preference.

introduction heuristic (NOISE) to break symmetry, resulting in (c). This heuristic helps increase parallelism by distributing instructions to different clusters. Then, we run *critical path strengthening* (PATH), which increases the weight of the instructions in the critical path (*i.e.,* instructions 23, 25, 26, etc.) in the first cluster (d). Then we run the *communication minimization* (COMM) and the *load balance* (LOAD) heuristics, resulting in (e). These heuristics lead to several changes: the first few instructions are pushed out of the first cluster, and groups of instructions start to assemble in specific clusters (*e.g.,* instructions 19, 20, 21, and 22 in cluster 3).

Next, we run PLACE and PLACEPROP, which bias instructions using information from preplaced nodes. The result is shown in (f). The pass causes a lot of disturbances: preplaced instructions strongly attract neighbors of preplaced instructions to their clusters. Observe how the group 19–21 is attracted to cluster 4. Finally we run *communication minimization* (COMM) another time. The final schedule is shown in (g).

Convergent scheduling has the following features:

1. Its scheduling decisions are made *cooperatively* rather than *exclusively.*
2. The interface allows a pass to express confidence about its decisions. A pass 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 made by one pass. In the example, when we apply noise to (b), most nodes are initially moved away from the first cluster. Subsequently, however, nodes with strong ties to cluster one, such as nodes 1–6, are eventually moved back, while nodes without strong ties, such as node 0, remain away.
4. Most compilers allow only very limited exchange of information among passes. In contrast, the weight-based interface to convergent scheduling is very expressive.
5. The framework allows a heuristic to be applied multiple times, either independently or as part of an iterative process. This feature is useful to provide feedback between passes and to avoid pass 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. Passes for different heuristics are written independently, and the expressive, common interface reduces design complexity. This offers an easy way to retarget a compiler and to address peculiarities of the underlying architecture. If, for example, an architecture is able to exploit auto-increment on memory-access with a specific instruction, one pass could try to keep together memory-accesses and increments, so that the scheduler will find them together and will be able to exploit the advanced instructions.

3. Convergent Interface

Convergent scheduling operates on individual scheduling units, which may be basic blocks, traces [10], hyperblocks [11] etc. It stores preferences in a three dimensional matrix $W_{i,c,t}$, where i spans over all instructions in the scheduling unit, c over the clusters in the architecture, and t over time. We allocate as many time slots as the critical-path length.

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

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

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

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

Given an instruction i, we define the following:[1]

$$\begin{aligned}
\text{preferred_time(i)} &= \arg\max \left\{ t : \sum_{c} W_{i,t,c} \right\} \\
\text{preferred_cluster(i)} &= \arg\max \left\{ c : \sum_{t} W_{i,t,c} \right\} \\
\text{runnerup_cluster(i)} &= \arg\max \left\{ \begin{array}{l} c : \sum_t W_{i,t,c}; \\ c \neq \text{preferred_cluster(i)} \end{array} \right\} \\
\text{confidence(i)} &= \frac{\sum_t W_{i,t,\text{preferred_cluster(i)}}}{\sum_t W_{i,t,\text{runnerup_cluster(i)}}}
\end{aligned}$$

The preferred values are those that maximize the sum of the preferences over time and clusters, while the confidence of an instruction measures how confident the convergent scheduler is about its current spatial assignment. It is computed as the ratio of the weights of the top two clusters. The following basic operations are available on weights:

- Any weight $W_{i,t,c}$ can be increased/decreased by a constant, or multiplied by a factor.
- The preference matrix of one instruction can be combined to that of another instruction, with a relative weight w ($0 \leq w \leq 1$). This way we *propagate* high-confidence decisions to neighboring instructions. We never perform this full operation because it is expensive. Instead, we only do this along the space dimension, or only within a small range along the time dimension.

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

- The system incrementally keeps track of the sums of the weights over both space and time, so that they can be determined in $O(1)$ time. It also memorizes the preferred time and preferred cluster of each instruction.
- The preferences can be normalized to guarantee our invariants; the normalization simply performs:

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

1. Given an expression to be maximized within a range, *max* returns the maximum value reached by the expression, while *argmax* is the value of the variable that maximizes it. For instance $\max\{x \in R : x - x^2\}$ is 1/4, while $\arg\max\{x \in R : x - x^2\}$ is 1/2, because for the value $x = 1/2$, we reach the maximum 1/4.

4. Collection of Heuristics

This section presents a collection of heuristics we have implemented for convergent scheduling. Each heuristic attempts to address a single constraint and only communicates with other heuristics via the weight matrix. There are no restrictions on the order or the number of times each heuristic is applied. In our first experiments, heuristics and their order were manually tuned, with a tedious trial-and-error process. In Section 5, we describe a more systematic approach to ordering passes, based on Genetic Programming.

Whenever necessary, we run normalization at the end of every pass to ensure the invariants described in Section 3. This step is implicit in the description below.

Initial Time Assignment (INITTIME). Instructions 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 and, for any instruction, l_p is the length of the longest path from the top of the graph to it (latency of predecessor chain), and l_s is the longest path from it 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. A pass similar to this one can address the fact that some instructions cannot be scheduled in all clusters in some architectures, simply by squashing the weights for the unfeasible clusters.

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

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

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

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

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

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

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

Critical Path Strengthening (PATH). This pass tries to keep all the instructions on a critical path (CP) in the same cluster. If instructions in the paths have bias for a particular cluster, the path is moved to that cluster. Otherwise the least loaded cluster is selected. If different portions of the paths have strong bias toward different clusters (*e.g.,* when there are two or more preplaced instructions on the path), the critical path is broken in two or

more pieces and kept locally close to the relevant home clusters. Let $cc(i)$ be the chosen cluster for the CP.

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

Communication Minimization (COMM). This pass reduces communication load by increasing the weight for an instruction to be in the same clusters where most of neighbors (successors and predecessors in the dependence graph) are. This is done by summing the weights of all the neighbors in a specific cluster, and using that to skew weights in the correct direction. We have also implemented a variant of this that considers *grand-parents* and *grand-children*, and we usually run it together with COMM.

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

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, because it solidifies the current preference mappings. Let t_i and c_i be the preferred time and cluster for instruction i:

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

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

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

Load Balance (LOAD). This pass performs load balancing across clusters. Each weight on a cluster is divided by the total load on that cluster, computed as the sum of the preferences of all instructions:

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

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

To perform the dual goals of instruction distribution without excessive communication, instructions on a level are partitioned into bins. Initially, the bin B_c for each cluster c contains instructions whose preferred cluster is c, and whose confidence is greater than a threshold (2.0). Then, we perform the following:

```
LevelDistribute: int l, int g
  I_l = Instruction i | level(i) = l
  foreach Cluster c
    I_l = I_l − B_c
  I_g = {i | i ∈ I_l; distance(i, find_closest_bin(i)) > g}
  while I_l ≠ φ
    B = round_robin_next_bin()
    i_closest = arg max{i ∈ I_g : distance(i, B)}
    B = B ∪ i_closest
    I_l = I_l − i_closest
    Update I_g
```

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

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

Path Propagation (PATHPROP). This pass selects high confidence instructions and propagates their convergent matrices along a path. The confidence threshold t is an input parameter. Let i_h be the selected confident instruction. The following code propagates i_h along a downward path. A similar function, which visits predecessors, propagates i_h along an upward path.

```
find i | i ∈ successor(i_h); confidence(i) < confidence(i_h)
while (i ≠ nil)
  for each (c, t), W_{i,t,c} ← 0.5W_{i,t,c} + 0.5W_{i_h,t,c}
  find i_n | i_n ∈ successor(i); confidence(i_n) < confidence(i_h)
  i ← i_n
```

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

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

5. Genetic Programming

From one generation to the next, architectures in the same processor family may have extremely different internal organizations, and thus have unique compilation needs. We have therefore developed a machine-learning tool to automatically customize our convergent scheduler to any given architecture. The tool generates a sequence of passes from those described above.

Of the many available machine-learning techniques, we chose to employ Genetic Programming (GP) because its attributes fit the needs of our application. Like many other evolutionary algorithms, it is based on the thesis that a computational version of fitness-based selection, reproductive inheritance and blind variation acting upon a population will

⟨*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`'
| '`LOAD`' | '`LEVEL`' | '`PATHPROP`' | '`EMPHCP`'
| '`BEST`' | '`PLACE`' | '`PLACEPROP`' | '`FIRST`'

Table 1: Grammar for genome s-expressions.

Variable	True if
Is imbalanced	the difference in load between the most and the least loaded cluster is larger than 1/*numcluster*
Is fat	the number of independent critical paths is larger than the number of tiles
Is within CPL	the number of instructions in the block is smaller than the number of tiles times the critical path length
Is placement bad	the number of *unplaced* instructions is more than half the number of instructions in the block

Table 2: Variables used by our system. Their values can change after each pass.

lead the individuals in subsequent generations to adapt toward better performance in their environment.

GP's attractive features include its ability to explore high-dimensional spaces, its high scalability (it can run effectively on a distributed cluster of workstations), and the fact that its solutions are human-readable, 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 (or equivalently, as LISP *s-expressions*). In our case, the parse trees represent a sequence of conditionally executed passes. Table 1 shows the grammar we use to describe pass orders. The <*variable*> expression is used to extract pertinent information about the status of the schedule, and the shape of the block under analysis. This introspection allows the scheduler to run different passes based on the state of the schedule. Table 2 shows the four variables used by our system.

The algorithm starts by creating an initial *population* of 200 random expressions. It then compiles and runs each of the benchmarks in our training set for each individual in the population. Each individual is then assigned a *fitness* based on how fast each of the associated programs in the *training set* executes. In our case, the fitness is simply

the average speedup for the benchmarks in the training set, compared to the hand-tuned sequence used in the previous section. We also reward *parsimony* by giving preference to the shorter of two otherwise equivalently fit sequences.

The weakest individuals (20%) are discarded, and replaced with new individuals: half of them completely randomly, the other half created via the crossover operator from the fittest individuals. To guard against stagnant populations, GP often uses mutation. One possible mutation simply replaces a randomly chosen subtree with a new random expression. The GP algorithm halts when a user-defined number of iterations (40, in our case) has been reached.

6. Results: Basic (Hand-tuned) Convergent Scheduling

We have implemented convergent scheduling in two systems: the Raw architecture [1] and the Chorus clustered VLIW infrastructure developed at MIT [12].

Experimental Environment. The Raw machine 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 addition to a traditional, wormhole dynamic network, Raw has a programmable, compiler-controlled *static network* that can be used to route scalar values between the register file/ALUs on different tiles (for details, please refer to [6]). Latency on the static network is three cycles for two neighboring tiles; each additional hop takes one extra cycle of latency.

RawCC, the Raw compiler, takes a sequential C or Fortran program and parallelizes it across Raw tiles. It is built on top of the MachSUIF intermediate representation [13]. RawCC divides each input program into one or more scheduling traces. For each trace, RawCC constructs the data precedence graph and performs space-time scheduling on each graph. Then, it applies a traditional register allocator to the code on each tile.

The Chorus clustered VLIW system is a flexible compiler/simulator environment, which can simulate many different configurations of clustered VLIW machines. The configuration we used most is a 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.

The Chorus compiler shares with RawCC the same high level structure. Like RawCC, it is implemented on top of MachSUIF. It first performs space-time scheduling, followed by traditional single-cluster register allocation [14]. Both RawCC and the Chorus compiler employ congruence transformation and analysis to increase and analyze the predictability of memory references [5]. 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.

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

Table 3: Hand-tuned sequence of heuristics used by the convergent scheduler for (a) the Raw machine and (b) clustered VLIW.

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.[2] On Chorus, all values that are live across multiple scheduling regions are mapped to the first cluster.

Architecture-specific Heuristics. Table 3 lists the heuristics used by the convergent scheduler for Raw and Chorus. These orders were hand-tuned to the architectures, with a trial-and-error process. 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, and (2) the temporal assignment of each instruction.

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

Benchmarks. Our sources of benchmarks include the Raw benchmark suite (jacobi, life) [15], Nasa7 of Spec92 (cholesky, vpenta, and mxm), and Spec95 (tomcatv, fpppp-kernel). Fpppp-kernel is the inner loop of fpppp 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. For technical limitations of our compilers, some benchmarks were not available for Chorus or RAW.

Performance Comparisons. We compared our results with the baseline RawCC and Chorus compilers. Table 4 compares the performance of convergent scheduling to RawCC

2. RawCC does use SSA (Single Static Assignment) renaming to eliminate false dependences, which in turn reduces these preplacement constraints.

	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 4: RawCC speedup. Speedup is relative to performance on one tile.

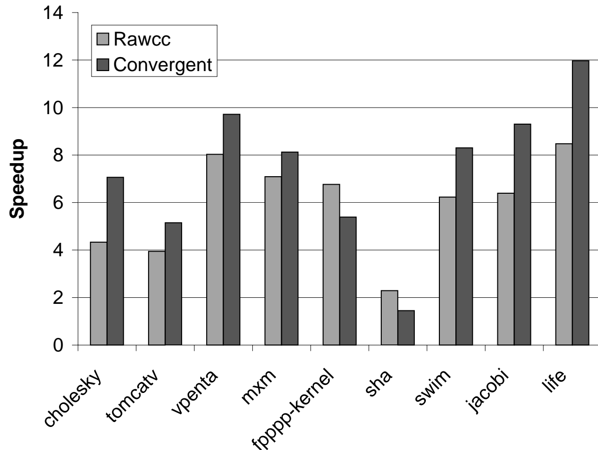


Figure 4: Performance comparisons between RawCC and Convergent scheduling on a 16-tile Raw machine. Speedup is relative to performance on one tile.

for two to 16 tiles. Figure 4 plots the same data for 16 tiles. Results show that convergent scheduling consistently outperforms baseline RawCC for all tile configurations for most of the benchmarks, with an average improvement of 21% for 16 tiles.

Many of our benchmarks are dense matrix code with preplaced memory instructions from congruence analysis. For these benchmarks, convergent scheduling always outperforms baseline RawCC. The reason is that convergent scheduling is able to actively take advantage of preplacement information to guide the placement of other instructions. This information leads 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 speedups on these benchmarks requires finding and exploiting very fine-grained parallelism. Our level distribution pass has been less efficient in this regard than the clustering counterpart in RawCC — we expect that integrating a clustering pass to convergent scheduling will address this problem.

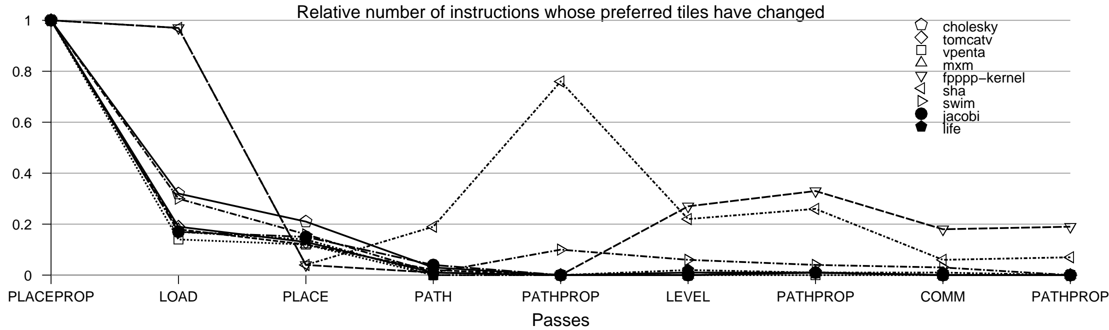


Figure 5: Convergence of spatial assignments on Raw.

Figure 5 shows the relative number of instructions whose preferred tiles are changed by each convergent pass on Raw. The plots measure static instruction counts, and they exclude passes that only modify temporal preferences. For benchmarks with useful preplacement information, the convergent scheduler is able to converge to good solutions quickly, by propagating the preplacement information and using the load balancing heuristic. In contrast, preplacement provides little useful information for fpppp-kernel and sha. These benchmarks thus require other critical paths, parallelism, and communication heuristics to converge to good assignments.

Some pass orderings could take the scheduling to local minima, but this did not happen in our experiments. Some central passes, i.e. PATHPROP or LEVEL, can move a large part of instructions in order to reach a better solution. Nonetheless, a final agreement is reached: in all cases, the last PATHPROP does not change many instructions.

Figure 6 compares the performance of convergent scheduling to two existing assignment/scheduling techniques for clustered VLIW: UAS [8] and PCC [9]. We augment each existing algorithm with preplacement information. For UAS, we modify the CPSC heuristic described in the original paper to give the highest priority to the home cluster of preplaced instructions. For PCC, the algorithm for estimating schedule lengths and communication costs properly accounts for preplacement information, by modeling the extra costs incurred by the clustered VLIW machine for a non-local memory access. Convergent scheduling outperforms UAS and PCC by 14% and 28%, respectively, on a 4-cluster VLIW machine. Like in Raw, the convergent scheduler is able to use preplacement information to find good natural partitions for our dense matrix benchmarks. Figure 7 shows the relative number of static instructions whose preferred tiles are changed by each convergent pass on Chorus. Passes that only modify temporal preferences are excluded.

Compile-time Scalability. We examined the scalability of convergent scheduling. Scalability is important because there is an increasing need for instruction assignment and scheduling algorithms to handle larger blocks. In fact, to extract the amount of instruction-level parallelism needed by modern and future microprocessors, compilers need to work

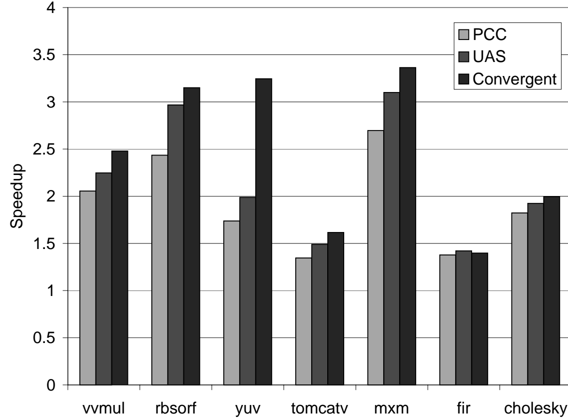


Figure 6: Performance comparisons between PCC, UAS, and Convergent scheduling on a 4-cluster VLIW. Speedup is relative to a single-cluster machine.

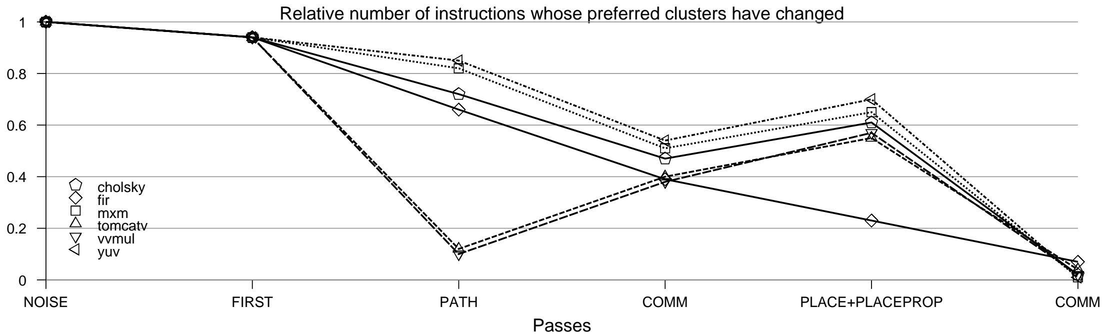


Figure 7: Convergence of spatial assignments on Chorus.

on more advanced block representations, such as hyperblocks and treegions, with a larger scheduling scope.

Figure 8 compares the compile-time of convergent scheduling with that of UAS and PCC on Chorus. Both convergent scheduling and PCC use an independent list scheduler after instruction assignment — our measurements include time spent in the scheduler. The figure shows that convergent scheduling and UAS take about the same amount of time. They both scale considerably better than PCC. We note that PCC is highly sensitive to the number of components it initially divides the instructions into. Compile-time can be dramatically reduced if the number of components is kept small. However, we find that for our benchmarks, reducing the number of components also results in much poorer assignment quality.

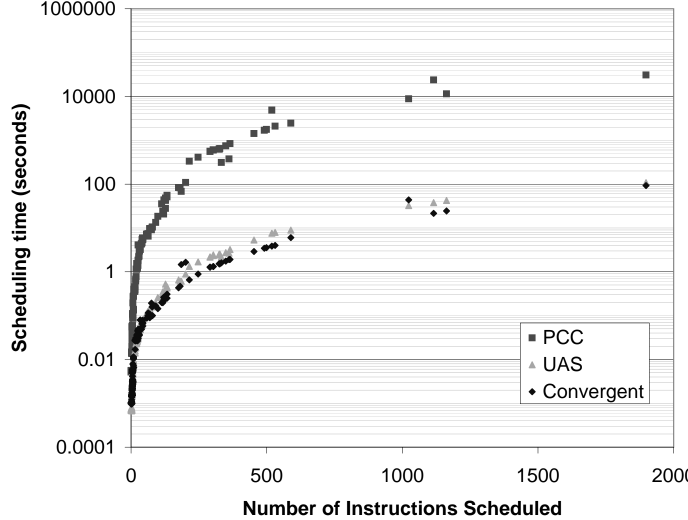


Figure 8: Comparison of compile-time vs input size for algorithms on Chorus.

7. Results: Genetic Programming

We tested our Genetic Programming framework on our clustered VLIW infrastructure, and we compared it with PCC [9] and UAS [8]. To verify the way our evolutionary system is able to learn to exploit new architectures, we run our experiments on three new configurations, which are modifications of the baseline used in the previous section.

Baseline (4cl). This is the configuration used in the previous experiments. 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 though 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.

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, thus creating a basic broadcasting capability. Because this model has limited bandwidth, the space-time scheduler must be more conservative in splitting computation across clusters.

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.

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).

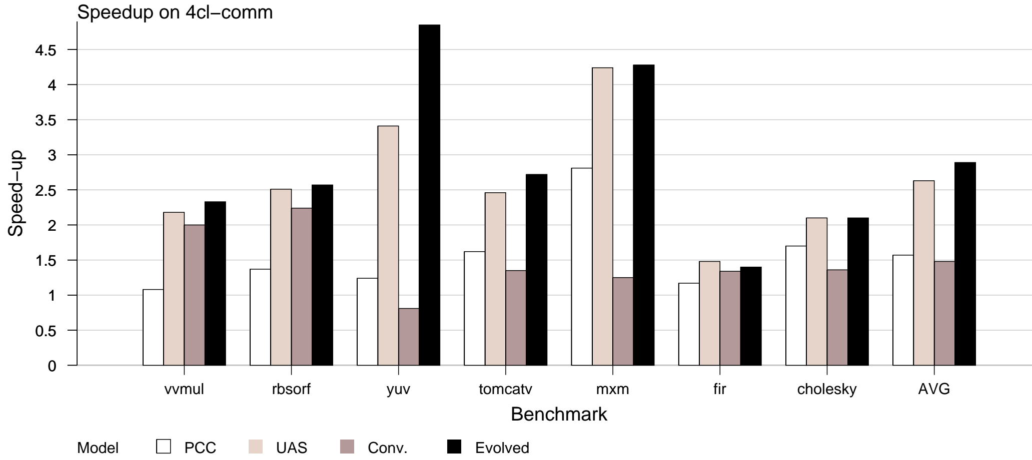


Figure 9: Speedup on 4cl-comm compared with 1-cluster convergent scheduling (original sequence). CONV is the baseline sequence (hand-tuned in our previous work), and EVOLVED is the performance of the sequence evolved for this architecture.

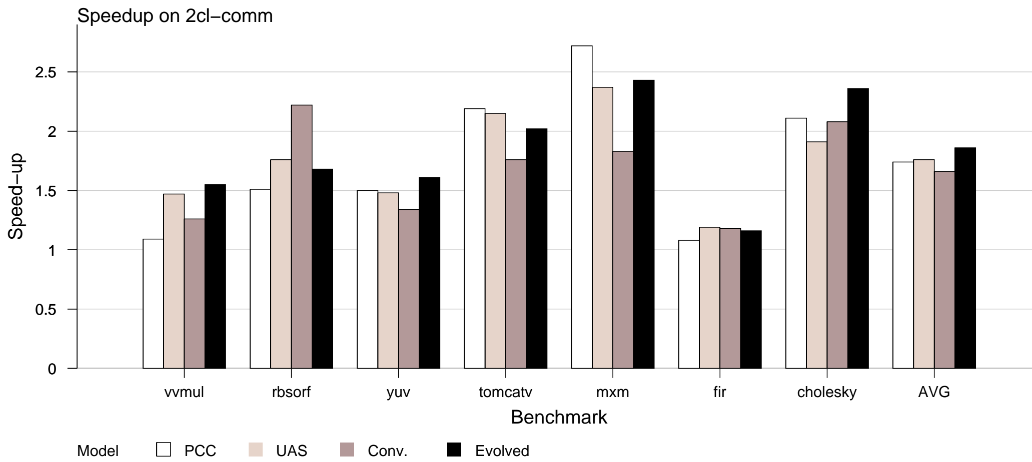


Figure 10: Speedup on 2cl-comm.

For each of these four architectures, we evolved an application-independent sequence of passes: our fitness function reward the pass orderings that performs well on all the benchmarks.[3] Results are shown in Figures 9, 10, and 11. The evolved sequence (*Evolved*

3. As said before, the fitness measures the average speed-up of the benchmarks compiled with a given sequence.

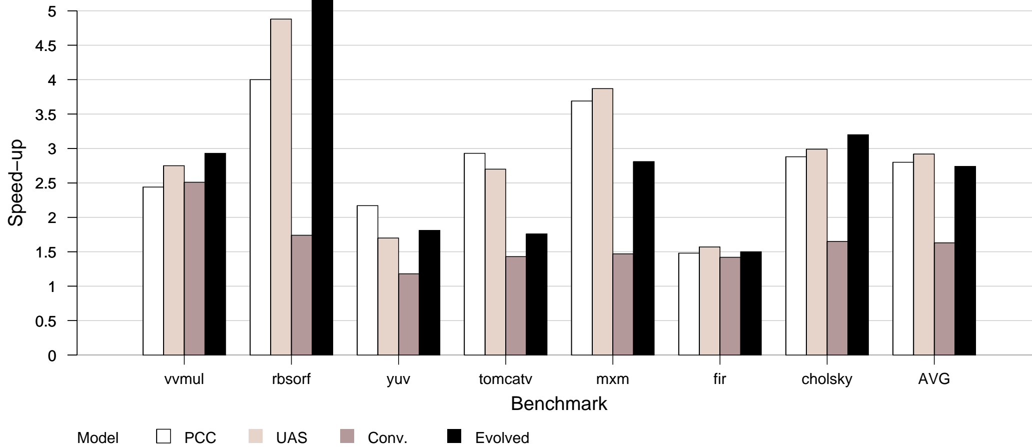


Figure 11: Speedup on 4cl-regs.

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 5: Results of cross validation (speed-up). The highlighted numbers refer to the benchmark excluded in each test.

in the figures) outperformed the initial sequence (*Conv.* in the figures), which was hand-tuned for the baseline architecture, of 95%, 12% and 68% in the three architectures.

The evolved convergent scheduler outperforms UAS and PCC, except in the case of limited registers, where performance is lower by 6% and 2% respectively. We are investigating new passes that address this aspect. Also, the original hand-tuned sequence is over-tuned for the baseline architecture, and it is generally outperformed by UAS and PCC in the other configurations.

Leave-one-out Cross Validation. We also tested the robustness of our system by using leave-one-out cross validation. The evolution was re-run excluding one of the seven benchmarks, and the resulting evolved pass ordering was tested again on the excluded benchmark. The seven cross-validation evolutions reached results very similar (within 6%) to the full evolution, for the excluded benchmarks too (see Table 5).

With these initial experiments, we verified that convergent scheduling is well suited to a set of different architectures. An improved sequence of passes can be found running our evolutionary framework on 20 dual Pentium 4 machines, in less than 40 hours. Once found, it can be used as the core of an architecture-specific application-independent compiler.

8. Related Work

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

Many compilers for spatial architectures address the three problems separately. Much research has focused on novel ways to do cluster assignment, coupled with traditional list scheduling and register allocation methods. The pioneering work in cluster assignment is BUG [4]. BUG uses a two-phase algorithm. First, the algorithm traverses a dependence graph bottom-up to propagate information about preplaced instructions. Then, it traverses the graph top-down and greedily maps each instruction to the cluster that can execute it earliest. The Multiflow compiler uses a variant of BUG [16], without the support for preplaced instructions.

PCC is an iterative assignment approach based on partial components [9]. It builds partial components by visiting the data dependence graph bottom-up, critical-path first. The maximum size of a component is limited by a parameter, ϕ_{th}. It uses simple heuristics to select a value for ϕ_{th} that balances the trade-off between performance and compile-time, although the exact method is not discussed in detail in the original paper. The components are initially assigned to clusters based on simple load balancing and communication criteria. The assignments are subsequently improved through iterative descent, by checking whether moving a sub-component to another cluster improves the schedule. RawCC leverages techniques developed for multiprocessor task graph scheduling [6]. Assignment is performed in three steps: *clustering* groups instructions that have little parallelism; *merging* reduces the number of clusters through merging; *placement* maps clusters to tiles. During placement, RawCC also handles constraints from preplaced instructions.

In [17], the solution differs from the above approaches in the ordering of passes. It performs scheduling before assignment. The assignment pass uses a min-cut algorithm adapted from circuit partitioning that tries to minimize communication. This algorithm, however, does not directly attempt to optimize the execution length of input DAGs.

To avoid pass ordering problems, recent works have proposed combined solutions. Leupers describes an iterative combined approach, based on simulated annealing, to perform scheduling and partitioning on a VLIW DSP [18]. UAS performs assignment and scheduling together by integrating assignment into a cycle-driven list scheduler [8]. CARS performs all three tasks — assignment, scheduling, and register allocation — in one step, by integrating both assignment and register allocation into a modified cycle-driven list scheduler [19].

Lerner proposes an interesting interface to different passes based on *graph replacement* [20]. His approach enables independently designed data-flow passes to be composed and run together. The composed pass is able to achieve the precision of iterating independent passes, but without the compile-time cost of iteration. In all these approaches,

however, every decision is irrevocable and final. In contrast, convergent scheduling provides a general framework that allows decisions to be postponed or reversed.

Pass ordering issues on clustered architectures is a relatively new area; a more classical pass ordering problem occurs in scalar optimizations. Some approaches in those areas share similar goals and features with convergent scheduling. Cooper et al. used a Genetic Algorithm solution to evolve the order of passes [21]. Their approach finds good general solutions, and it performs even better when the evolution is applied independently on each benchmark. Our research extends this work in many significant ways. First, our learning representation allows for conditional execution of passes, while theirs does not. Second, we simultaneously train on multiple benchmarks to create general-purpose solutions. Finally, our Genetic Programming representation allows us to search solutions of variable length: by rewarding parsimony, we can find very compact solutions, when possible, or more complex solutions when simple sequences are not suitable. On the contrary, Cooper's approach, based on Genetic Algorithm, is limited to solutions of a fixed size. We believe the convergent scheduling solution space is more interesting than that of an ordinary back-end. The symmetry and unselfishness of convergent scheduling passes implies an interesting and immense solution space.

A recent work by Kulkarni et al. [22] describes new techniques to improve the efficiency of Genetic Algorithms. Their results might reduce the time needed to search our optimized pass orderings.

9. Conclusions

Time-to-market pressures make it difficult to effectively target next generation processors. Convergent scheduling's simple interface alleviates such constraints by facilitating rapid prototyping of passes. As shown, convergent scheduling simplifies compiler design by providing a common interface by which independent passes can cooperatively exchange their *beliefs* about the space-time schedule. Also, any pass can be run freely before or after any other, so simplifying the cost of deploying and testing a new compiler. Nonetheless, we show that our convergent scheduling compiler is still fast and effective.

Our work shows also how machine-learning techniques can be used to automatically search the pass-order solution space. Our Genetic Programming technique allowed us to easily re-target new architectures, by discovering more effective sequences. A compiler can be tuned to a new architecture by running our evolutionary system for convergent scheduling. Cross validation tests show that performance improvement is not limited to the benchmarks on which the sequence is trained: the one-time evolution will return a general purpose compiler tuned to the new architecture. This process can be repeated for any new machine, adding, if needed, some architecture-specific passes to the framework.

The results shows that convergent scheduling is effective on a variety of architectures, offers quick compilation (comparable to other modern compilers), adapts to different architectural configuration. We believe convergent scheduling is a novel, effective approach to compiling for modern machines.

Acknowledgements

This work extends and integrates our results, previously published as [23, 24]. We want to thank Shane Swenson, Martin Martin, and Una-May O'Reilly for their contribution to this project.

References

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

[2] R. Nagarajan, K. Sankaralingam, D. Burger, and S. Keckler, "A Design Space Evaluation of Grid Processor Architectures," 2001.

[3] H.-S. Kim and J. 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.

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

[5] S. Larsen and S. Amarasinghe, "Increasing and Detecting Memory Address Congruence," in *Proceedings of 11th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, (Charlottesville, VA), September 2002.

[6] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe, "Space-Time Scheduling of Instruction-Level Parallelism on a RAW Machine," in *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, CA), pp. 46–57, Oct. 1998.

[7] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen, "Combining Register Allocation and Instruction Scheduling," Tech. Rep. CS-TN-95-22, 1995.

[8] E. Ozer, S. Banerjia, and T. M. Conte, "Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures," in *International Symposium on Microarchitecture*, pp. 308–315, 1998.

[9] G. Desoli, "Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach," Tech. Rep. HPL-98-13, Hewlett Packard Laboratories, January 1998.

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

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

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

[13] M. D. Smith, "Machine SUIF," in *National Compiler Infrastructure Tutorial at PLDI 2000*, June 2000. http://www.eecs.harvard.edu/hube.

[14] L. George and A. W. Appel, "Iterated Register Coalescing," in *ACM Transactions on Programming Languages and Systems*, vol. 18, 1996.

[15] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal, "The RAW Benchmark Suite: Computation Structures for General Purpose Computing," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, (Napa Valley, CA), April 1997.

[16] 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*, pp. 51–142, Jan. 1993.

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

[18] R. Leupers, "Instruction Scheduling for Clustered VLIW DSPs," in *International Conference on Parallel Architecture and Compilation Techniques*, (Philadelphia, PA, USA), Oct. 2000.

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

[20] S. Lerner, D. Grove, and C. Chambers, "Composing Dataflow Analyses and Transformations," in *The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, (Portland, Oregon), 2002.

[21] K. Cooper, D. Subramanian, and L. Torczon, "Adaptive Optimizing Compilers for the 21st Century," *Journal of Supercomputing*, vol. 23, pp. 7–22, August 2002.

[22] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones, "Fast Searches for Effective Optimization Phase Sequences," *ACM SIGPLAN*, vol. 39, pp. 171–182, May 2004.

[23] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe, "Convergent Scheduling," in *Proceedings of 35th International Symposium on Microarchitecture (MICRO)*, (Istanbul, Turkey), 2002.

[24] D. Puppin, M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Adapting Convergent Scheduling Using Machine-Learning," in *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, (College Station, TX), 2003.