Adapting Convergent Scheduling Using Machine-Learning

Diego Puppin[1], Mark Stephenson[2], Saman[2] Amarasinghe[2], Martin Martin[2], and Una-May O'Reilly[2]

[1] Institute for Information Science and Technologies
ISTI - CNR, Pisa, Italy
diego.puppin@alum.mit.edu
[2] Massachusetts Institute of Technology
{mstephen, saman}@cag.lcs.mit.edu
{mcm, unamay}@ai.mit.edu

Abstract. Convergent scheduling is a general framework for instruction scheduling and cluster assignment for parallel, clustered architectures. A convergent scheduler is composed of many independent passes, each of which implements a specific compiler heuristic. Each of the passes shares a common interface, which allows them to be run multiple times, and in any order. Because of this, a convergent scheduler is presented with a vast number of legal pass orderings.
In this work, we use machine-learning techniques to automatically search for good orderings. We do so by evolving, through genetic programming, s-expressions that describe a particular pass sequence. Our system has the flexibility to create dynamic sequences where the ordering of the passes is predicated upon characteristics of the program being compiled. In particular, we implemented a few tests on the present state of the code being compiled. We are able to find improved sequences for a range of clustered architectures. These sequences were tested with cross-validation, and generally outperform Desoli's PCC and UAS.

1 Introduction

Instruction scheduling on modern microprocessors is an increasingly 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. 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 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 [1]. The third, and most common approach, is to address the constraints one at a time in a sequence of passes. This approach however, introduces pass ordering problems, as decisions made by early passes are based on partial information and can adversely affect the quality of decisions made by subsequent passes.

Convergent Scheduling [2, 3] alleviates pass ordering problems by spreading scheduling decisions over the entire compilation. Each pass makes soft decisions about instruction placement: it asserts its preference of instruction placement, but does not impose a hard schedule on subsequent passes. All passes in the convergent scheduler share a common interface: the input and output to each one is a collection of spatial and temporal preferences of instructions: a pass operates by modifying these data. As the scheduler applies the passes in succession, the preference distribution will converge to a final schedule that incorporates the preferences of all the constraints and heuristics.

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. In our previous work [3], we tediously hand-tuned the pass order. This paper builds upon it by using 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. 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 speedups ranging from 12% to 95% over our previous work, and generally outperform UAS [4] and PCC [5].

The remainder of the paper is organized as follows. Section 2 describes Genetic Programming, the machine-learning technique we use to explore the phase-order solution space. We describe our infrastructure and methodology in Section 3. Section 4 quickly describes the set of available heuristics. Section 5 follows with a description of the experimental results. Section 6 discusses related work, and finally, Section 7 concludes. Because of limited space, we refer you to [2, 3] for architecture and implementation details related to convergent scheduling.

2 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 pressure, it is necessary to automate as much of the design process as possible. In our first 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

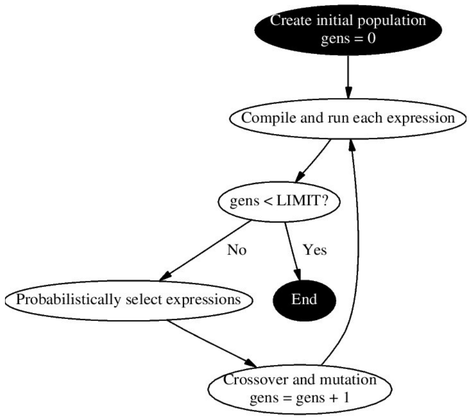
Create initial population
gens = 0
Compile and run each expression
gens < LIMIT?
No
Yes
Probabilistically select expressions
End
Crossover and mutation
gens = gens + 1

Fig. 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.

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 passes from those described in section 4. This section describes genetic programming (GP), the machine-learning technique that our tool uses.

Of the many available learning techniques, we chose to employ genetic programming because its attributes fit the needs of our application. GP [6] is one example of evolutionary algorithm (EA). The thesis behind evolutionary computation is that a computational version of fitness-based selection, reproductive inheritance and blind variation acting upon a population will lead the individuals in subsequent generations to adapt toward better performance in their environment.

In the general GP framework, individuals are represented as parse trees (or equivalently, as lisp expressions) [6]. In our case, the parse trees represent a sequence of conditionally executed passes.The result of each subexpression is either a convergent scheduling phase, or a sequence of phases. Our system evaluates an individual in a pre-order traversal of the tree.

Table 1 shows the grammar we use to describe phase 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 schedule state. The four variables that our system considers are shown in Table 2.

Table 1. Grammar for genome s-expressions. <variable> returns the value computed by our tests on the graph and the current schedule.

```
⟨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'
```

Figure 1 shows the general flow of genetic programming. The algorithm starts by creating an initial *population* of random parse trees. 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* execute. In our case, the fitness is simply the average speedup (compared to the sequence used in previous work) over all the benchmarks in the training set.

The fittest individuals are chosen for crossover, the GP analogy of sexual reproduction. Crossover begins by choosing two well-fit individuals. Our system then clones the selected individuals, chooses a random subexpression in each of them, and swaps them. The net result is two new individuals, composed of building blocks from two fit parents.

To guard against stagnant populations, GP often uses mutation. Mutations simply replace a randomly chosen subtree with a new random expression. For details on the mutation operators we implemented, see [7, p. 242]. In our implementation, the GP algorithm halts when a user-defined number of iterations has been reached.

We conclude this section by noting some of GP's attractive features. First, it is capable of exploring high-dimensional spaces. It is also highly scalable, highly parallel and can run effectively on a distributed cluster of workstations. In addition, its solutions are human-readable, compared with other algorithms (e.g. neural networks) where the solution is embedded in a very complex state space.

3 Infrastructure and Methodology

This section describes our compilation framework as well as the methodology we use to collect results. We begin by describing the GP parameters we used to

Table 2. The variables used by our system. Their values are updated during compilation.

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

train the convergent scheduler. This section concludes with a description of our experimental compiler and VLIW simulator.

3.1 GP Parameters

We wrapped the GP framework depicted in Figure 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*. All experiments maintain a population of 200 individuals, initially randomly chosen. After every generation we discard the weakest 20% of the population, and replace them with new individuals. New individuals are created to replace the discarded portion of the population. Of these new phase orderings, half of them are complelety random, and the remainder are created via the crossover operator described in the last section. 5% of the individuals created via crossover are subject to mutation. Finally, we run each experiment for 40 generations.

Fitness is measured as the average speed-up (over all the benchmarks in our training suite) when compared against the phase ordering that we used in our previous work [3]. We also reward *parsimony* by giving preference to the shorter of two otherwise equivalently fit sequences.

3.2 Compiler Flow and Simulation Environment

Our compilation process begins in the SUIF front-end [8]. In addition to performing alignment analysis [9], the front-end carries out traditional optimizations such as loop unrolling, constant propagation, copy propagation, and dead code elimination.

Our *Chours* VLIW back-end follows [10]. Written using MachSUIF [11], the back-end allows us to easily vary the number of clusters, functional units, and registers in the target architecture. Instruction latencies, memory access latencies, and inter-cluster communication latencies are also configurable. The convergent scheduler uses such information, combined with data from alignment analysis, to generate effective code. Similarly, our register allocator must know the number of registers in each cluster.

The result of the compilation process is a compiled simulator that we use to collect performance numbers. The simulator accurately models the latency of each functional unit. We assume that all functional units are fully pipelined. Furthermore, the simulator enforces lock-step execution. Thus, if a memory instruction misses in the cache, all clusters will stall. The memory system is runtime configurable so we can easily isolate the performance of various memory topologies. In total, the back-end comprises nine compiler passes and a simulation library.

The four target architectures on which we experimented are described below.

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 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).

4 Available Passes

In this section, we describe quickly the passes used in our experimental framework. Passes are divided into time heuristics, passes for placement and critical path, for communication and load balancing, and register allocation. The miscellaneous passes help the convergence by breaking symmetry and strengthening the current assignment. For implementation details, we refer the reader to [2, 3].

4.1 Time heuristics

Initital time assignment (INITTIME) initializes the weight matrix by squeezing to 0 all the time slots that are unfeasible for a particular instruction. If the distance to the farthest root of the data-depedency graph is t, the preference for that instruction to be scheduled a cycle earlier than t is set to 0. The distance to the leaf is similarly used.

Dependence enforcement (DEP) verifies that no instruction is scheduled before an instruction on which it depends. This is done by reducing the preference for early time slots in the dependent instruction.

Functional units (FUNC) reduces the preference for overloaded time-slots, i.e. slots for which the load is higher than the number of available functional units.

Emphasize critical path distance (EMPHCP) tries to schedule every instruction at the time indicated by its level, i.e. the distance from roots and leaves.

4.2 Placement and Critical Path

Push to first cluster (FIRST) gives instructions a slight bias to the first cluster, where our compiler guarantees the presence of all alive registers at the end of each block (so, less communication is needed for instructions in the first cluster).

Preplacement (PLACE) increases, for preplaced instructions (see [9]), the preference for their home cluster.

Preplacement propagation (PLACEPROP) propagates the information about preplacement to neighbors in the data dependence graph. The preference for each cluster decreases with the distance (in the dependence graph) from the closest preplaced instruction in that cluster.

Critical path strengthening (PATH) identifies one critical path in the schedule, and tries to keep it together in the least loaded cluster or in the home cluster of its preplaced instructions.

Path propagation (PATHPROP) identifies high-confidence instructions, and propagates their preferences to the neighbors in the critical path.

Create clusters (CLUSTER) creates small instruction clusters using the Partial Component Clustering [5], and then allocates them to clusters trying to minimize communication. This is useful when the preplacement information is poor.

4.3 Communication and Load Balancing

Communication minimization (COMM) tries to minimize communication by keeping in the same cluster instructions that are neighbors in the dependence graph.

Parallelism for successors (SUCC) exploits the broadcast feature of some of our VLIW configurations by distributing across clusters the children of an instruction which is already communicating data on the bus. The other instructions can snoop the value, so no extra communications will be needed.

Load balance(LOAD) reduces the preferences for the cluster that has the highest preferences so far.

Level distribute (LEVEL) tries to put in different clusters the instructions that are in the same level (distance from roots and leaves) if they do not communicate.

4.4 Register allocation

Break edges (EDGES) tries to reduce register pressure by breaking the data dependence edges that cross any specific time t in the schedule (if there are more edges than the available registers). This is done by reducing the preferences of the instructions in the edges to be scheduled around t.

Reduce parallelism (SEQUENTIAL) emphasizes the sequential order of instructions in the basic block. This reduces parallelism and register pressure due to values with long life-span.

4.5 Miscellaneous

Noise introduction (NOISE) adds noise to the distribution to break symmetry in subsequent choices.

Assignment strengthening (BEST) boosts the highest preference in the schedule, so far.

5 Results

In this section, we compare the performance of convergent scheduling to two existing assignment/scheduling techniques for clustered VLIW architectures: UAS [4] and PCC [5]. 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. It does so by modeling the extra costs incurred by the clustered VLIW machine for a non-local memory access.

For simplicity, in the following, we will refer to the sequence `(SEQ (PassA) (PassB))` simply as `(PassA) (PassB)`, removing `SEQ`: when no variables are used, genomes reduce to a linear sequence of passes. Also, 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.

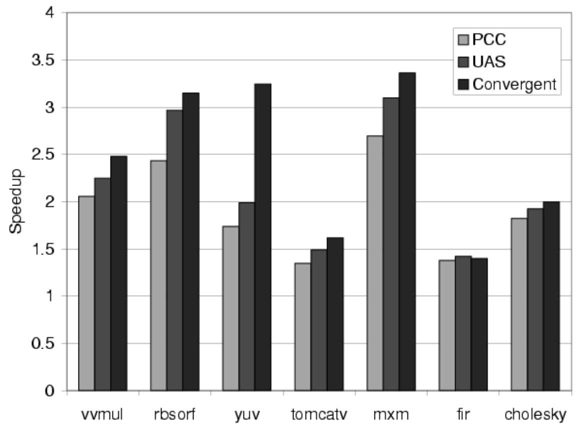

Fig. 2. Performance comparisons between PCC, UAS, and Convergent scheduling on a four-cluster VLIW architecture. Speedup is relative to a single-cluster machine.

5.1 Baseline (4cl)

The baseline sequence was hand-tuned in our initial work with convergent scheduling. For the baseline architecture, our compiler used the following sequence:

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

As shown in Figure 2, convergent scheduling outperforms UAS and PCC by 14% and 28%, respectively, on a four-clustered VLIW machine. Convergent scheduling is able to use preplacement information to find good natural partitions for our dense matrix benchmarks.

5.2 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 4 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 5.5), we limit our evolution to 40 generations.

The evolved sequence is more conservative in communication. `(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;

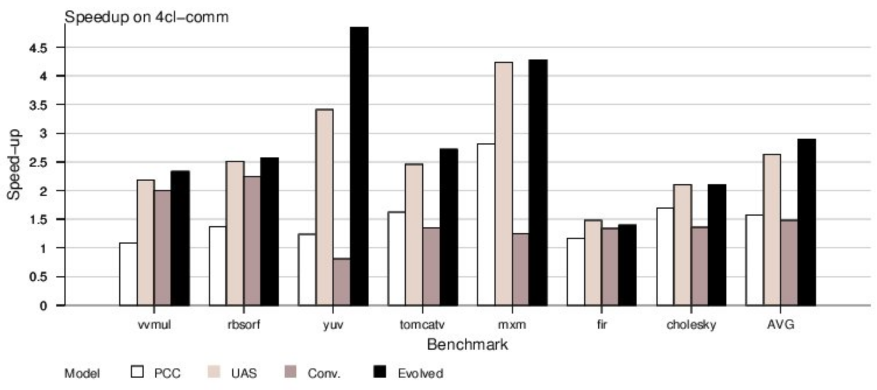

Fig. 3. Speedup on 4cl-comm compared with 1-cluster convergent scheduling (original sequence). In the graph, conv. is the baseline sequence, evolved is the new sequence for this architecture.

`(func)` reduces peaks on overloaded clusters, which could lead to high amounts of localized communication. Also, the `(comm)` pass is run twice, in order to limit the total communication load.

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

The plot in Figure 3 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 modeling of communication done by our implementation of the internal simplified scheduler (see [5]).

5.3 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 *shaking* up the schedule.

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

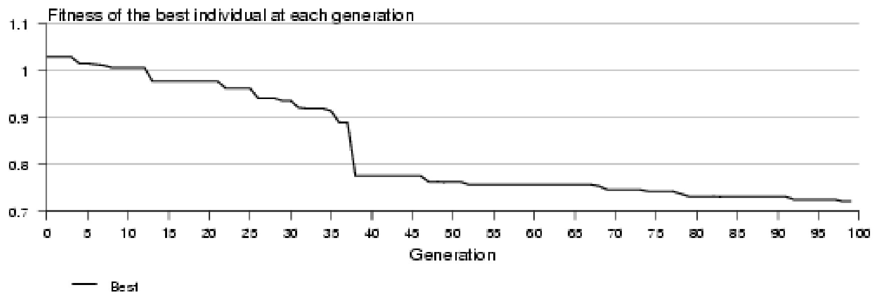

Fig. 4. Completion time for the set of benchmarks for the fittest individual, during evolution on 4cl-comm.

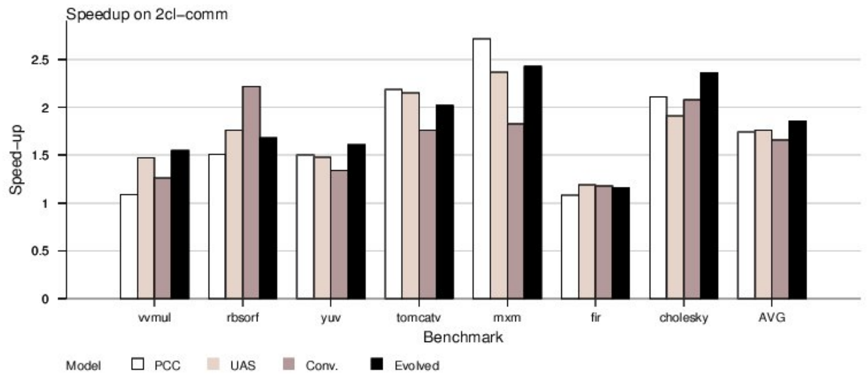

Fig. 5. Speedup on 2cl-comm.

5.4 Limited Registers (4cl-regs)

Figure 6 shows the performance of the evolved sequence when compared with our baseline and our reference. We 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%. We believe this is due to the need for new expressive heuristics for register allocation. Future work will investigate this.

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

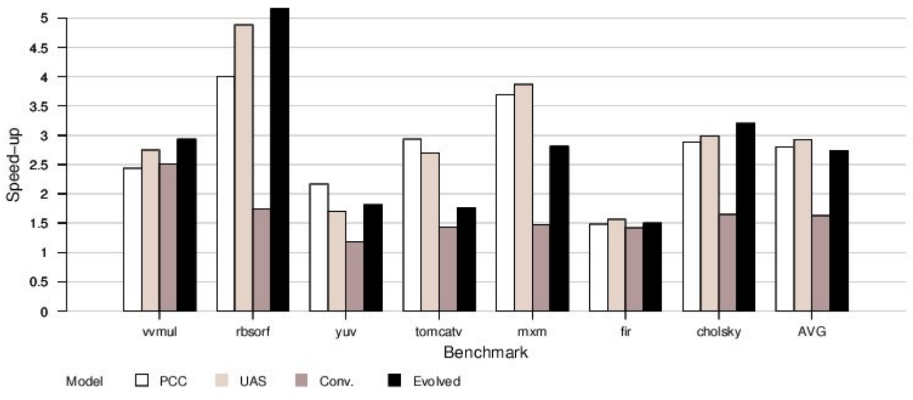

Fig. 6. Speedup on 4cl-regs.

Table 3. The sequence evolved in our cross-validation tests.

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)`

5.5 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 benchmark. In Table 4, the results are shown as speed-up compared with a one-cluster architecture. The seven cross-validation evolutions reached results very similar to the full evolution, for the excluded benchmarks too. In particular, the sequences evolved excluding one benchmark still outperform, on average, the comparison compilers, UAS and PCC.

The seven evolved sequences (in table 3) are all similar: `(func)` is the most important pass for this architecture.

Table 4. Results of cross validation, speed-up compared with 1-cluster architecture. The highlighted numbers refer to the performance of the excluded benchmark, when using the evolved sequence.

	Excluded benchmark							
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

5.6 Summary of Results

We verified that convergent scheduling is well suited to a set of different architectures. Running on 20 dual-processor 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.

6 Related work

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 [12]. 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 use the application-specific sequences to hand-craft a general-purpose solution. 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 [13]. 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 [14]. 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.

7 Conclusion

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. In addition, an architecture-specific pass is not as susceptible to bad decisions made by previously run passes as in ordinary compilers.

Because the scheduler's framework allows passes to be run in any order, there are countless legal phase orders to consider. This paper 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 this paper, 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 reasons to believe that this is a promising approach. Future work will explore this in greater detail.

In closing, 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.

Acknowledgements

We want thank Shane Swenson and Walt Lee for their contribution. This work has been partially supported by the Italian National Research Council (CNR) FIRB project GRID.it "Enabling platforms for high-performance computational grids oriented to scalable virtual organizations," by a grant from DARPA (PCA F29601-04-2-0166), an award from NSF (CISE EIA-0071841) and fellowships from the Singapore-MIT Alliance and the MIT Oxygen Project.

References

1. Motwani, R., Palem, K.V., Sarkar, V., Reyen, S.: Combining register allocation and instruction scheduling. Technical Report CS-TN-95-22 (1995)
2. Puppin, D.: Convergent scheduling: A flexible and extensible scheduling framework for clustered vliw architectures. Master's thesis, Massachusetts Institute of Technology (2002)
3. Lee, W., Puppin, D., Swenson, S., Amarasinghe, S.: Convergent scheduling. In: Proceedings of the 35th International Symposium on Microarchitecture, Istanbul, Turkey (2002)
4. Ozer, E., Banerjia, S., Conte, T.M.: Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In: International Symposium on Microarchitecture. (1998) 308–315
5. Desoli, G.: Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach. Technical Report HPL-98-13, Hewlett Packard Laboratories (1998)
6. Koza, J.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press (1992)
7. Banzhaf, W., Nordin, P., Keller, R., Francone, F.: Genetic Programming : An Introduction : On the Automatic Evolution of Computer Programs and Its Applications. Morgan Kaufmann (1998)
8. Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S., Hennessy, J.L.: SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. SIGPLAN **29** (1994) 31–37
9. Larsen, S., Amarasinghe, S.: Increasing and detecting memory address congruence. In: Proceedings of 11th International Conference on Parallel Architectures and Compilation Techniques (PACT), Charlottesville, VA (2002)
10. Maze, D.: Compilation infrastructure for vliw machines. Master's thesis, Massachusetts Institute of Technology (2001)
11. Smith, M.D.: Machine SUIF. In: National Compiler Infrastructure Tutorial at PLDI 2000. (2000) http://www.eecs.harvard.edu/hube.
12. Cooper, K.D., Schielke, P.J., Subramanian, D.: 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)
13. Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., Zorn, B.: Evidence-Based Static Branch Prediction Using Machine Learning. In: ACM Transactions on Programming Languages and Systems (ToPLaS-19). Volume 19. (1997)
14. Grewal, G.W., Wilson, C.T.: Mappping Reference Code to Irregular DSPs with the Retargetable, Optimizing Compiler COGEN(T). In: International Symposium on Microarchitecture. Volume 34. (2001) 192–202