# Adapting Convergent Scheduling Using Machine-Learning

Diego Puppin
*Institute for Information Science and Technologies*
*ISTI - CNR, Pisa, Italy*
*diego.puppin@alum.mit.edu*

Mark Stephenson, Saman Amarasinghe,
Martin Martin and Una-May O'Reilly
*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, where some passes can be executed conditionally if some conditions are found in the given code. 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. 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* 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 [6], 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 speedup ranging from 12% to 95% over previous work, and generally outperform UAS [7] and PCC [3].

The rest of this paper is structured as follows: in the next section, our genetic programming and compiler frameworks are described. Section 3 reports our results, which are com-

pared with related work in section 4. Finally, section 5 concludes.

## 2. Experimental Infrastructure

### 2.1. 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 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 in [6].

Of the many available machine-learning techniques, we chose to employ genetic programming (GP) because its attributes fit the needs of our application. As 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 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 state of the schedule. The variables that our system considers currently are shown in table 2.

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 [6]. 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 random, 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.

| | | |
|---|---|---|
| ⟨*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' \| 'FIRST' |
| | | \| 'SEQUENTIAL' \| 'CLUSTER' \| 'EMPHCP' |

**Table 1. Grammar for genome s-expressions.**

### 2.2. Compiler Flow and Simulation Environment

Our compilation process begins in the SUIF front-end [8]. In addition to performing alignment analysis [5], the front-end carries out traditional optimizations such as loop unrolling, constant propagation, copy propagation, and dead code elimination.

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 run-time 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.

## 3. Results

We compared the performance of convergent scheduling with two existing assignment/scheduling techniques for clustered VLIW architectures: PCC [3] and UAS [7]. We augmented each existing algorithm with preplacement information (see [6]).

For each of the three tested architectures (*four and two clusters with limited communication bandwidth* (4cl-comm, 2cl-comm), and *four clusters with reduced number of registers* (4cl-regs)), we tried to evolve an application-

independent conditional sequence of passes for the compiler.

The evolved sequence outperformed the initial sequence, hand-tuned for a four-cluster architecture with full communication bandwidth, of 95%, 12% and 68% in the three architectures (see figures). The 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.

We also tested the robustness of our system by using leave-one-out cross validation. The evolution was rerun excluding one of the seven benchmarks, and the result was tested again on the excluded benchmark. The seven cross-validation evolutions reached results very similar (within 3%) to the full evolution, for the excluded benchmarks too.

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.

We noticed that 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: running extra passes is usually not detrimental to the final result. So, our bias for shorter genomes (parsimony pressure) penalizes sequences with extra tests. Nevertheless, we still believe in the potential of this approach, and leave further exploration to future work.

## 4. Related work

Many researchers have used machine-learning techniques to solve hard compiling problems. Nonetheless, our work offers many novel contributions. To the best of our knowledge, this is the first time a compiler is evolved, in an unsupervised way, independently from specific applications. Previous works [4, 2] used evolution on a per-application basis to reach compact, high-performance code on DSP and embedded processors. Calder et al.[1] used supervised learning to improve the branch prediction heuristics of their compiler. Our problem demands an unsupervised method since optimal compiler sequences are not known.

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

| 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.**

Because the scheduler's framework allows passes to be run in any order, there are countless legal pass orders to consider. This work shows how machine-learning techniques could 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. Also, cross validation showed that performance improvement is not limited to the benchmarks on which the sequence was trained.

To conclude, we would like to highlight that the general framework of convergent scheduling can be extended also for other scheduling/assignment problems. For instance, it can be used for the task of scheduling on distributed architectures, where the computation grain is coarser. The general infrastructure can be used to map computation on machines, instead of clusters within a processor. Convergent scheduling can be adapted to new metrics of communication and distance. We leave this to future work.

## Acknowledgements

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."

## References


[1] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-Based Static Branch Prediction Using Machine Learning. In *ToPLaS-19*, 1997.

[2] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *LCTES*, 1999.

[3] G. Desoli. Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach. Technical Report HPL-98-13, HP Labs, January 1998.

[4] G. W. Grewal and C. T. Wilson. Mappping Reference Code to Irregular DSPs with the Retargetable, Optimizing Compiler COGEN(T). In *MICRO-34*, pages 192–202, 2001.

[5] S. Larsen and S. Amarasinghe. Increasing and detecting memory address congruence. In *PACT-2002*, Charlottesville, VA, September 2002.

[6] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent scheduling. In *MICRO-35*, 2002.

[7] E. Ozer, S. Banerjia, and T. M. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *MICRO-31*, pages 308–315.

[8] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN*, 29(12):31–37, Dec. 1994.

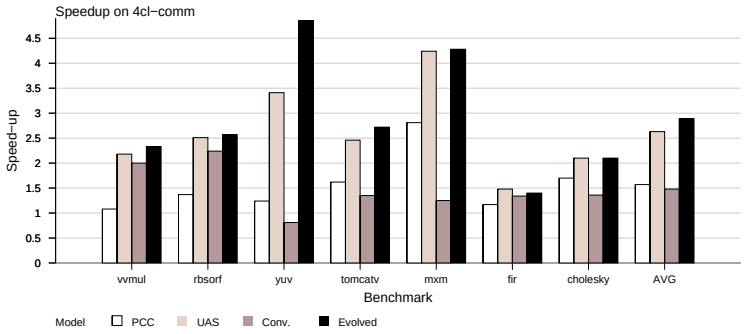


**Figure 1. Speedup on 4cl-comm compared with 1-cluster convergent scheduling (original sequence). CONV is the baseline sequence (hand-tuned in our previous work), and EVOLVED is the performance of the sequence evolved for this architecture.**

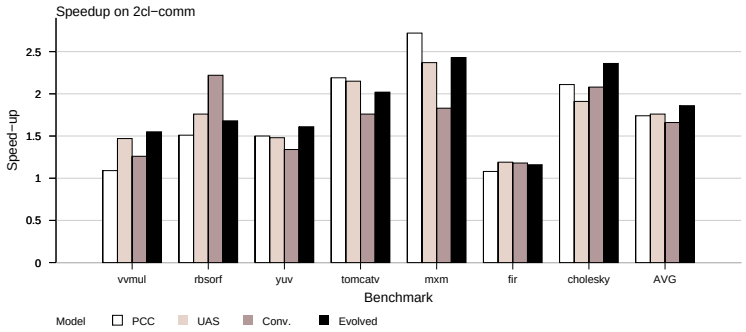


**Figure 2. Speedup on 2cl-comm.**

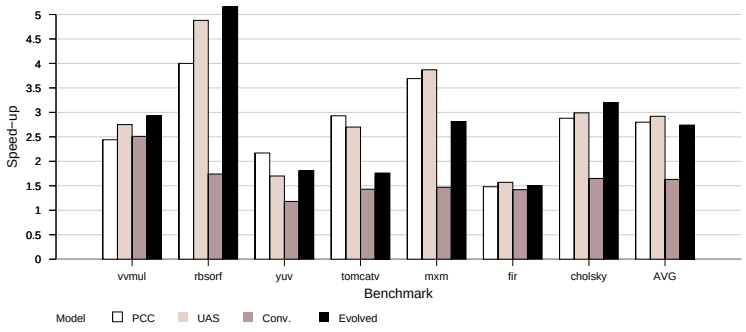


**Figure 3. Speedup on 4cl-regs.**