Learning to Schedule Straight-Line Code

Eliot Moss, Paul Utgoff, John Cavazos
Doina Precup, Darko Stefanović
Dept. of Comp. Sci., Univ. of Mass.
Amherst, MA 01003

Carla Brodley, David Scheeff
Sch. of Elec. and Comp. Eng.
Purdue University
W. Lafayette, IN 47907

Abstract

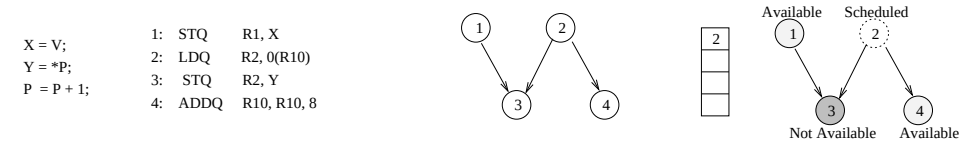
Program execution speed on modern computers is sensitive, by a factor of two or more, to the order in which instructions are presented to the processor. To realize potential execution efficiency, an optimizing compiler must employ a heuristic algorithm for instruction scheduling. Such algorithms are painstakingly hand-crafted, which is expensive and time-consuming. We show how to cast the instruction scheduling problem as a learning task, obtaining the heuristic scheduling algorithm automatically. Our focus is the narrower problem of scheduling straight-line code (also called *basic blocks* of instructions). Our empirical results show that just a few features are adequate for quite good performance at this task for a real modern processor, and that any of several supervised learning methods perform nearly optimally with respect to the features used.

1 Introduction

Modern computer architectures provide semantics of execution equivalent to sequential execution of instructions one at a time. However, to achieve higher execution efficiency, they employ a high degree of internal parallelism. Because individual instruction execution times vary, depending on when an instruction's inputs are available, when its computing resources are available, and when it is presented, overall execution time can vary widely. Based on just the semantics of instructions, a sequence of instructions usually has many permutations that are easily shown to have equivalent meaning—but they may have considerably different execution time. Compiler writers therefore include algorithms to schedule instructions to achieve low execution time. Currently, such algorithms are hand-crafted for each compiler and target processor. We apply learning so that the scheduling algorithm is constructed automatically.

Our focus is *local instruction scheduling*, i.e., ordering instructions within a *basic block*. A basic block is a straight-line sequence of code, with a conditional or unconditional branch instruction at the end. The scheduler should find optimal, or good, orderings of the instructions prior to the branch. It is safe to assume that the compiler has produced a semantically correct sequence of instructions for each basic block. We consider only *reorderings* of each sequence

(not more general rewritings), and only those reorderings that *cannot affect the semantics*. The semantics of interest are captured by *dependences* of pairs of instructions. Specifically, instruction I_j depends on (must follow) instruction I_i if: it follows I_i in the input block and has one or more of the following dependences on I_i: (a) I_j uses a register used by I_i and at least one of them writes the register (condition codes, if any, are treated as a register); (b) I_j accesses a memory location that may be the same as one accessed by I_i, and at least one of them writes the location. From the input total order of instructions, one can thus build a *dependence DAG*, usually a partial (not a total) order, that represents all the semantics essential for scheduling the instructions of a basic block. Figure 1 gives a sample basic block and its DAG. The task of scheduling is to find a least-cost total order of each block's DAG.



(a) C Code (b) Instruction Sequence to be Scheduled (c) Dependence Dag of Instructions (d) Partial Schedule

Figure 1: Example basic block code, DAG, and partial schedule

2 Learning to Schedule

The learning task is to produce a scheduling procedure to use in the performance task of scheduling instructions of basic blocks. One needs to transform the partial order of instructions into a total order that will execute as efficiently as possible, assuming that all memory references "hit" in the caches. We consider the class of schedulers that repeatedly select the apparent best of those instructions that could be scheduled next, proceeding from the beginning of the block to the end; this *greedy* approach should be practical for everyday use.

Because the scheduler selects the apparent best from those instructions that could be selected next, the learning task consists of learning to make this selection well. Hence, the notion of 'apparent best instruction' needs to be acquired. The process of selecting the best of the alternatives is like finding the maximum of a list of numbers. One keeps in hand the current best, and proceeds with pairwise comparisons, always keeping the better of the two. One can view this as learning a relation over triples (P, I_i, I_j), where P is the partial schedule (the total order of what has been scheduled, and the partial order remaining), and I is the set of instructions from which the selection is to be made. Those triples that belong to the relation define pairwise preferences in which the first instruction is considered preferable to the second. Each triple that does not belong to the relation represents a pair in which the first instruction is not better than the second.

One must choose a representation in which to state the relation, create a process by which correct examples and counter-examples of the relation can be inferred, and modify the expression of the relation as needed. Let us consider these steps in greater detail.

2.1 Representation of Scheduling Preference

The representation used here takes the form of a logical relation, in which known examples and counter-examples of the relation are provided as triples. It is then a matter of constructing or revising an expression that evaluates to TRUE if (P, I_i, I_j) is a member of the relation, and FALSE if it is not. If (P, I_i, I_j) is considered to be a member of the relation, then it is safe to infer that (P, I_j, I_i) is not a member.

For any representation of preference, one needs to represent features of a candidate instruction and of the partial schedule. There is some art in picking useful features for a state. The method

used here was to consider the features used in a scheduler (called DEC below) supplied by the processor vendor, and to think carefully about those and other features that should indicate predictive instruction characteristics or important aspects of the partial schedule.

2.2 Inferring Examples and Counter-Examples

One would like to produce a preference relation consistent with the examples and counter-examples that have been inferred, and that generalizes well to triples that have not been seen. A variety of methods exist for learning and generalizing from examples, several of which are tested in the experiments below. Of interest here is how to infer the examples and counter-examples needed to drive the generalization process.

The focus here is on supervised learning (reinforcement learning is mentioned later), in which one provides a process that produces correctly labeled examples and counter-examples of the preference relation. For the instruction-scheduling task, it is possible to search for an optimal schedule for blocks of ten or fewer instructions. From an optimal schedule, one can infer the correct preferences that would have been needed to produce that optimal schedule when selecting the best instruction from a set of candidates, as described above. It may well be that there is more than one optimal schedule, so it is important only to infer a preference for a pair of instructions when the first can produce some schedule better than any the second can.

One should be concerned whether training on preference pairs from optimally scheduled small blocks is effective, a question the experiments address. It is worth noting that for programs studied below, 92% of the basic blocks are of this small size, and the average block size is 4.9 instructions. On the other hand, larger blocks are executed more often, and thus have disproportionate impact on program execution time. One could learn from larger blocks by using a high quality scheduler that is not necessarily optimal. However, the objective is to be able to learn to schedule basic blocks well for new architectures, so a useful learning method should not depend on any pre-existing solution. Of course there may be some utility in trying to improve on an existing scheduler, but that is not the longer-term goal here. Instead, we would like to be able to construct a scheduler with high confidence that it produces good schedules.

2.3 Updating the Preference Relation

A variety of learning algorithms can be brought to bear on the task of updating the expression of the preference relation. We consider four methods here.

The first is the decision tree induction program ITI (Utgoff, Berkman & Clouse, in press). Each triple that is an example of the relation is translated into a vector of feature values, as described in more detail below. Some of the features pertain to the current partial schedule, and others pertain to the pair of candidate instructions. The vector is then labeled as an example of the relation. For the same pair of instructions, a second triple is inferred, with the two instructions reversed. The feature vector for the triple is constructed as before, and labeled as a counter-example of the relation. The decision tree induction program then constructs a tree that can be used to predict whether a candidate triple is a member of the relation.

The second method is table lookup (TLU), using a table indexed by the feature values of a triple. The table has one cell for every possible combination of feature values, with integer valued features suitably discretized. Each cell records the number of positive and negative instances from a training set that map to that cell. The table lookup function returns the most frequently seen value associated with the corresponding cell. It is useful to know that the data set used is large and generally covers all possible table cells with multiple instances. Thus, table lookup is "unbiased" and one would expect it to give the best predictions possible for the chosen features, assuming the statistics of the training and test sets are consistent.

The third method is the ELF function approximator (Utgoff & Precup, 1997), which constructs

additional features (much like a hidden unit) as necessary while it updates its representation of the function that it is learning. The function is represented by two layers of mapping. The first layer maps the features of the triple, which must be boolean for ELF, to a set of boolean feature values. The second layer maps those features to a single scalar value by combining them linearly with a vector of real-valued coefficients called weights. Though the second layer is linear in the instruction features, the boolean features are nonlinear in the instruction features.

Finally, the fourth method considered is a feed-forward artificial neural network (NN) (Rumelhart, Hinton & Williams, 1986). Our particular network uses scaled conjugate gradient descent in its back-propagation, which gives results comparable to back-propagation with momentum, but converges much faster. Our configuration uses 10 hidden units.

3 Empirical Results

We aimed to answer the following questions: Can we schedule as well as hand-crafted algorithms in production compilers? Can we schedule as well as the best hand-crafted algorithms? How close can we come to optimal schedules? The first two questions we answer with comparisons of program execution times, as predicted from simulations of individual basic blocks (multiplied by the number of executions of the blocks as measured in sample program runs). This measure seems fair for local instruction scheduling, since it omits other execution time factors being ignored. Ultimately one would deal with these factors, but they would cloud the issues for the present enterprise. Answering the third question is harder, since it is infeasible to generate optimal schedules for long blocks. We offer a partial answer by measuring the number of optimal choices made within small blocks.

To proceed, we selected a computer architecture implementation and a standard suite of benchmark programs (SPEC95) compiled for that architecture. We extracted basic blocks from the compiled programs and used them for training, testing, and evaluation as described below.

3.1 Architecture and Benchmarks

We chose the Digital Alpha (Sites, 1992) as our architecture for the instruction scheduling problem. When introduced it was the fastest scalar processor available, and from a dependence analysis and scheduling standpoint its instruction set is simple. The 21064 *implementation* of the instruction set (DEC, 1992) is interestingly complex, having two dissimilar pipelines and the ability to issue two instructions per cycle (also called *dual issue*) if a complicated collection of conditions hold. Instructions take from one to many tens of cycles to execute.

SPEC95 is a standard benchmark commonly used to evaluate CPU execution time and the impact of compiler optimizations. It consists of 18 programs, 10 written in FORTRAN and tending to use floating point calculations heavily, and 8 written in C and focusing more on integers, character strings, and pointer manipulations. These were compiled with the vendor's compiler, set at the highest level of optimization offered, which includes compile- or link-time instruction scheduling. We call these the *Orig* schedules for the blocks. The resulting collection has 447,127 basic blocks, composed of 2,205,466 instructions.

3.2 Simulator, Schedulers, and Features

Researchers at Digital made publicly available a simulator for basic blocks for the 21064, which will indicate how many cycles a given block requires for execution, assuming all memory references hit in the caches and translation look-aside buffers, and no resources are busy when the basic block starts execution. When presenting a basic block one can also request that the simulator apply a heuristic greedy scheduling algorithm. We call this scheduler *DEC*.

By examining the DEC scheduler, applying intuition, and considering the results of various

preliminary experiments, we settled on using the features of Table 1 for learning. The mapping from triples to feature vectors is: *odd:* a single boolean 0 or 1; *wcp*, *e*, and *d:* the sign ($-$, 0, or $+$) of the value for I_j minus the value for I_i; *ic:* both instruction's values, expressed as 1 of 20 categories. For ELF and NN the categorical values for *ic*, as well as the signs, are mapped to a 1-of-*n* vector of bits, *n* being the number of distinct values.

Table 1: Features for Instructions and Partial Schedule

Heuristic Name	Heuristic Description	Intuition for Use
Odd Partial (odd)	Is the current number of instructions scheduled odd or even?	If TRUE, we're interested in scheduling instructions that can dual-issue with the previous instruction.
Instruction Class (ic)	The Alpha's instructions can be divided into equivalence classes with respect to timing properties.	The instructions in each class can be executed only in certain execution pipelines, etc.
Weighted Critical Path (wcp)	The height of the instruction in the DAG (the length of the longest chain of instructions dependent on this one), with edges weighted by expected latency of the result produced by the instruction	Instructions on longer critical paths should be scheduled first, since they affect the lower bound of the schedule cost.
Actual Dual (d)	Can the instruction dual-issue with the previous scheduled instruction?	If Odd Partial is TRUE, it is important that we find an instruction, if there is one, that can issue in the same cycle with the previous scheduled instruction.
Max Delay (e)	The earliest cycle when the instruction can begin to execute, relative to the current cycle; this takes into account any wait for inputs for functional units to become available	We want to schedule instructions that will have their data and functional unit available earliest.

This mapping of triples to feature values loses information. This does not affect learning much (as shown by preliminary experiments omitted here), but it reduces the size of the input space, and tends to improve both speed and quality of learning for some learning algorithms.

3.3 Experimental Procedures

From the 18 SPEC95 programs we extracted all basic blocks, and also determined, for sample runs of each program, the number of times each basic block was executed. For blocks having no more than ten instructions, we used exhaustive search of all possible schedules to (a) find instruction decision points with pairs of choices where one choice is optimal and the other is not, and (b) determine the best schedule cost attainable for either decision. Schedule costs are *always* as judged by the DEC simulator. This procedure produced over 13,000,000 distinct choice pairs, resulting in over 26,000,000 triples (given that swapping I_i and I_j creates a counter-example from an example and vice versa). We selected 1% of the choice pairs at random (always insuring we had matched example/counter-example triples).

For each learning scheme we performed an 18-fold cross-validation, holding out one program's blocks for independent testing. We evaluated both how often the trained scheduler made optimal decisions, and the simulated execution time of the resulting schedules. The execution time was computed as the sum of simulated basic block costs, weighted by execution frequency as observed in sample program runs, as described above.

To summarize the data, we use geometric means across the 18 runs of each scheduler. The geometric mean $g(x_1,\ldots,x_n)$ of $x_1,\ldots,x_n$ is $(x_1 \cdot \ldots \cdot x_n)^{1/n}$. It has the nice property that $g(x_1/y_1,\ldots,x_n/y_n) = g(x_1,\ldots,x_n)/g(y_1,\ldots,y_n)$, which makes it particularly meaningful for comparing performance measures via ratios. It can also be written as the anti-logarithm of the mean of the logarithms of the x_i; we use that to calculate confidence intervals using traditional measures over the logarithms of the values. In any case, geometric means are preferred for aggregating benchmark results across differing programs with varying execution times.

3.4 Results and Discussion

Our results appear in Table 2. For evaluations based on predicted program execution time, we compare with Orig. For evaluations based directly on the learning task, i.e., optimal choices, we compare with an optimal scheduler, but only over basic blocks no more than 10 instructions long. Other experiments indicate that the DEC scheduler almost always produces optimal schedules for such short blocks; we suspect it does well on longer blocks too.

Table 2: Experimental Results: Predicted Execution Time

	Relevant Blocks Only		All Blocks		Small Blocks
Sche-duler	cycles ($\times 10^9$)	ratio to Orig (95% conf. int.)	cycles ($\times 10^9$)	ratio to Orig (95% conf. int.)	% Optimal Choices
DEC	24.018	0.979 (0.969,0.989)	28.385	0.983 (0.975,0.992)	
TLU	24.338	0.992 (0.983,1.002)	28.710	0.995 (0.987,1.003)	98.1
ITI	24.395	0.995 (0.984,1.006)	28.758	0.996 (0.987,1.006)	98.2
NN	24.410	0.995 (0.983,1.007)	28.770	0.997 (0.986,1.008)	98.1
ELF	24.465	0.998 (0.985,1.010)	28.775	0.997 (0.988,1.006)	98.1
Orig	24.525	1.000 (1.000,1.000)	28.862	1.000 (1.000,1.000)	
Rand	31.292	1.276 (1.186,1.373)	36.207	1.254 (1.160,1.356)	

The results show that all supervised learning techniques produce schedules predicted to be better than the production compilers, but not as good as the DEC heuristic scheduler. This is a striking success, given the small number of features. As expected, table lookup performs the best of the learning techniques. Curiously, relative performance in terms of making optimal decisions does not correlate with relative performance in terms of producing good schedules. This appears to be because in each program a few blocks are executed very often, and thus contribute much to execution time, and large blocks are executed disproportionately often. Still, both measures of performance are quite good.

What about reinforcement learning? We ran experiments with temporal difference (TD) learning, some of which are described in (Scheeff, *et al.*, 1997) and the results are not as good. This problem appears to be tricky to cast in a form suitable for TD, because TD looks at candidate instructions in isolation, rather than in a preference setting. It is also hard to provide an adequate reward function and features predictive for the task at hand.

4 Related Work

Instruction scheduling is well-known and others have proposed many techniques. Also, optimal instruction scheduling for today's complex processors is NP-complete. We found two pieces of more closely related work. One is a patent (Tarsy & Woodard, 1994). From the patent's claims it appears that the inventors trained a simple perceptron by adjusting weights of some heuristics. They evaluate each weight setting by scheduling an entire benchmark suite, running the resulting programs, and using the resulting times to drive weight adjustments. This approach appears to us to be potentially very time-consuming. It has two advantages over our technique: in the learning process it uses measured execution times rather than predicted or simulated times, and it does not require a simulator. Being a patent, this work does not offer experimental results. The other related item is the application of genetic algorithms to tuning weights of heuristics used in a greedy scheduler (Beaty, S., Colcord, & Sweany, 1996). The authors showed that different hardware targets resulted in different learned weights, but they did not offer experimental evaluation of the quality of the resulting schedulers.

5 Conclusions and Outlook

While the results here do not demonstrate it, it was not easy to cast this problem in a form suitable for machine learning. However, once that form was accomplished, supervised learning produced quite good results on this practical problem—better than two vendor production compilers, as shown on a standard benchmark suite used for evaluating such optimizations. Thus the outlook for using machine learning in this application appears promising.

On the other hand, significant work remains. The current experiments are for a particular processor; can they be generalized to other processors? After all, one of the goals is to improve and speed processor design by enabling more rapid construction of optimizing compilers for proposed architectures. While we obtained good performance *predictions*, we did not report performance on a real processor. (More recently we obtained those results (Moss, *et al.*, 1997); ELF tied Orig for the best scheme.) This raises issues not only of faithfulness of the simulator to reality, but also of *global instruction scheduling*, i.e., across basic blocks, and of somewhat more general rewritings that allow more reorderings of instructions. From the perspective of learning, the broader context may make supervised learning impossible, because the search space will explode and preclude making judgments of optimal vs. suboptimal. Thus we will have to find ways to make reinforcement learning work better for this problem. A related issue is the difference between learning to make optimal decisions (on small blocks) and learning to schedule (all) blocks well. Another relevant issue is the cost not of the schedules, but of the schedulers: are these schedulers fast enough to use in production compilers? Again, this demands further experimental work. We do conclude, though, that the approach is promising enough to warrant these additional investigations.

Acknowledgments: We thank various people of Digital Equipment Corporation, for the DEC scheduler and the ATOM program instrumentation tool (Srivastava & Eustace, 1994), essential to this work. We also thank Sun Microsystems and Hewlett-Packard for their support.

References

Beaty, S., Colcord, S., & Sweany, P. (1996). Using genetic algorithms to fine-tune instruction-scheduling heuristics. In *Proc. of the Int' Conf. on Massively Parallel Computer Systems*.

Digital Equipment Corporation, (1992). *DECchip 21064-AA Microprocessor Hardware Reference Manual*, Maynard, MA, first edition, October 1992.

Haykin, S. (1994). *Neural networks: A comprehensive foundation*. New York, NY: Macmillan.

Moss, E., Cavazos, J., Stefanović, D., Utgoff, P., Precup, D., Scheeff, D., & Brodley, C. (1997). Learning Policies for Local Instruction Scheduling. Submitted for publication.

Rumelhart, D. E., Hinton, G. E., & Williams, R.J. (1986). Learning internal representations by error propagation. In Rumelhart & McClelland (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition*. Cambridge, MA: MIT Press.

Scheeff, D., Brodley, C., Moss, E., Cavazos, J., Stefanović, D. (1997). Applying Reinforcement Learning to Instruction Scheduling within Basic Blocks. Technical report.

Sites, R. (1992). *Alpha Architecture Reference Manual*. Digital Equip. Corp., Maynard, MA.

Srivastava, A. & Eustace, A. (1994). ATOM: A system for building customized program analysis tools. In *Proc. ACM SIGPLAN '94 Conf. on Prog. Lang. Design and Impl.*, 196–205.

Sutton, R. S. (1988). Learning to predict by the method of temporal differences. *Machine Learning, 3*, 9-44.

Tarsy, G. & Woodard, M. (1994). Method and apparatus for optimizing cost-based heuristic instruction schedulers. US Patent #5,367,687. Filed 7/7/93, granted 11/22/94.

Utgoff, P. E., Berkman, N. C., & Clouse, J. A. (in press). Decision tree induction based on efficient tree restructuring. *Machine Learning*.

Utgoff, P. E., & Precup, D. (1997). *Constructive function approximation*, (Technical Report 97-04), Amherst, MA: University of Massachusetts, Department of Computer Science.