S M A R T

Workshop on **S**tatistical and **M**achine Learning approaches applied to **Ar**chi**t**ectures and Compilation

Workshop Proceedings

January 28, 2007

Sofitel, Ghent, Belgium

Table of Contents

Track: Compilation

Tuning an Adaptive Compiler 2
Keith D. Cooper, Tim Harvey, and Jeff Sandoval

An Effective Local Search Algorithm for an Adaptive Compiler 7
Yi Guo, Devika Subramanian, and Keith D. Cooper

Basic-block Reordering Using Neural Networks 12
Xianhua Liu, Jiyu Zhang, Kun Liang, Yang Yang, and Xu Cheng

Black Box Approach for Selecting Optimization Options Using Budget-Limited Genetic Algorithms 27
Guy Bashkansky and Yaakov Yaari

Pruning the Optimization Search Space Using Architecture-aware Cost Models 43
Apan Qasem and Ken Kennedy

Building a Practical Iterative Interactive Compiler 53
Grigori Fursin and Albert Cohen

Track: Computer Architecture

Feasability of Combined Area and Performance Optimization for Superscalar Processors Using Searching 68
Sven von Haastregt and Peter M. W. Knijnenburg

On the Comparison of Regression Algorithms for Computer Architecture Performance Analysis of Software Applications 81
ElMoustapha Ould-Ahmed-Vall, James Woodlee, Charles Yount, and Kshitij Doshi

Branch Prediction with Bayesian Networks 96
Jeremy Singer, Gavin Brown, and Ian Watson

Integrated CPU and L2 cache Frequency/Voltage Scaling using Supervised Learning 113
Cosmin Rusu, Nevine AbouGhazaleh, Alexandre Ferreira, Ruibin Xu, Bruce Childers, Rami Melhem, and Daniel Mosse

Track: Compilation

Tuning an Adaptive Compiler

Keith D. Cooper, Tim Harvey, and Jeff Sandoval

Rice University, Houston, TX, 77005, USA
{keith,harv,jasandov}@cs.rice.edu

1 Introduction

Adaptive compilation is a technique for feedback-driven selection of program-specific or procedure-specific sequences of code optimizations [1]. Adaptive compilers find effective compilation sequences using search, machine learning, genetic algorithms, and limited enumeration [1–5]. A growing body of literature has established that no single sequence works well for all codes.

Active research in this area focuses on a number of problems, including finding better search techniques [4, 5], understanding the search spaces [1], and reducing the cost of evaluating distinct sequences [2]. This paper examines how the "best" sequences change when we add a new transformation to the search space. Specifically, we added a loop unroller to our pool of optimizations; it changed the "best" sequences and improved overall performance on a suite of benchmark codes.

1.1 Background

Our adaptive compiler focuses on the application of "scalar" optimizations to a low-level intermediate code [2]. Prior to this work, the compiler used 15 distinct transformations. It had no loop unroller, but included a pass, *peel*, that peels the first iteration out of each inner loop. This pass was intended to prepare the code for loop unswitching.

In our experiments, we noticed that the compiler often chose to apply peel repeatedly. For example, the five best sequences for the code `adpcm_coder` each use peel seven times; each 'p' indicates a use of peel.

pppppppczpc	pppppppclpc	pppppppclpd	pppppppclpg	pppppppclpm

Table 1. The five best sequences for `adpcm_coder` without unrolling

Because peel eliminates overhead from the first iteration of a loop, it produces a small improvement on each application. As Table 2 shows, peel is chosen quite often. It accounts for 26% of the passes invoked in the best 1% of sequences for our nine benchmark programs.

Where peeling reduces the overhead for the first iteration of the loop, unrolling reduces the overhead for the entire loop. Thus, a single application of unroll achieves a larger improvement than a single application of peel, or even several applications of peel. To see this effect, we added unroll to the fixed sequence against which we test the adaptive algorithms. With unrolling, we saw up to an 8% speedup; unfortunately, on average, it slowed the code down by 5.5%.

Benchmark	Loop-peel Frequency	Benchmark	Loop-peel Frequency
`adpcm_coder`	42.8%	`adpcm_decoder`	21.8%
`applu`	40.0%	`matrix300`	13.1%
`rkf45`	41.5%	`seval`	15.3%
`solve`	30.5%	`svd`	24.0%
`tomcatv`	4.8%	**average**	**26.0%**

Table 2. Frequency of peel in top 1% of sequences, by benchmark.

The variability of results from unrolling make it an excellent candidate for the adaptive compiler. Since a pass is only applied when profitable, we can add the unroller and let the search algorithms discover when to use it. Adding an unroller improved the quality of code that the compiler produced and changed both the best sequences found and the distribution of passes in those sequences.

This paper chronicles our experience adding a loop unroller to our adaptive compiler. We measured the compiler's behavior with and without the unroller. For sequences of 10 passes, the best sequences with unroll produce code that is 10% faster, on average, than those without unroll. Unroll often improves search efficiency, even though it almost doubles the search space size. The compiler usually finds a solution of a given quality more quickly with unroll than without it.

1.2 Loop Unrolling

Loop unrolling clones the body of a loop some number of times (the unroll factor) and adjusts the iteration count [6]. It reduces the number of induction-variable updates and backward branches in the loop. While unrolling is straightforward at the source-code level, our compiler uses an assembly-like intermediate code, where loops are harder to recognize. In this setting, the implementation of unrolling is more complex. Though the details are beyond the scope of this paper, our method combines cycle analysis in control-flow graphs [7] and induction-variable detection [8] in static-single-assignment graphs [9].

Qasem *et al.* argue that a phase-ordering framework such as ours is a poor place to choose transformation parameters like the unroll factor [3]. Selecting an integer parameter for the unroller poses different challenges than does sequence finding. Furthermore, a different unroll factor might make sense for each loop. To handle this issue, we constrain the behavior of the unroller. When applied, it unrolls every inner loop by a factor of four. The search can achieve larger unroll factors by applying the pass multiple times.

1.3 Hill Climber Framework

We performed this work using the impatient hill climber in our adaptive framework [2]. The hill climber starts at a randomly-selected sequence and examines its Hamming-1 neighbors in random order. If it finds a neighbor that executes fewer operations, it moves to that sequence and begins to examine its neighbors. When it can find no improvement, it declares the sequence a local minimum.

To limit search times, the hill climber is impatient; it examines a limited set of neighbors. If none of those sequences is an improvement, it restarts from a new sequence. Experience shows that the hill climber finds good sequences in five to ten descents [2].

Our previous work used sequences of 10 transformations drawn from a set of 15, a space of $15^{10} = 576,650,390,625$ points. Adding unroll enlarges the space to $16^{10} = 1,099,511,627,776$ points, nearly double the size. Though the new space is a strict superset of the old one, it is difficult to predict the effects of the change. We expected that, for programs with opportunities for unrolling, the hill climber would find better sequences, since nearly half of the sequences in the new space contain unroll. We also expected that the search would find "good" sequences faster, because unrolling takes better advantage of opportunities where peel is profitable. On the other hand, programs for which unroll produces no effect—or negative effects—may produce slower searches that return lower quality sequences if the hill climber wastes time evaluating unprofitable sequences.

2 Experimental Results

To determine how adding a transformation changes the system's behavior, we performed a series of experiments. All measurements show dynamic instruction counts for a simulated virtual machine. Prior experience suggests that we will see similar results with our SPARC and PowerPC backends. The simulator, however, produces stable, easily measured behavior.

2.1 Unroll in a Fixed Sequence

Figure 1 summarizes the impact of adding unroll to our compiler. For each benchmark, the leftmost bar shows the dynamic instruction count for code compiled with our standard fixed sequence [2]. The other measurements are normalized to the performance of the standard sequence. The second bar indicates the performance achieved by applying unroll before the standard sequence. Unroll improves performance up to 8% on `matrix300` and 2% to 6% on `seval` and `tomcatv`. Unfortunately, it degrades performance up to 24% on `applu`, `rkf45`, `solve`, and `svd`, mainly due to small iteration counts in many inner loops. Both `adpcm_coder` and `adpcm_decoder` show no change due to a subtle naming issue that prevents unroll from finding induction variables. Coalescing copies before unrolling would fix the problem; however, the adaptive search automatically finds the correct enabling transformations.

2.2 Unroll in an Adaptive Hill Climber

The two rightmost bars in Figure 1 show the best sequences found by the hill climber without unroll and with unroll. The hill climber used 20 restarts and examined at most 32 neighbors of each point. The hill climber performs well in the larger search space. Each program, except `solve`, shows improvement of 1% to 25% from the addition of unroll. The average improvement from adding unroll, including the degradation on `solve`, is 10%.

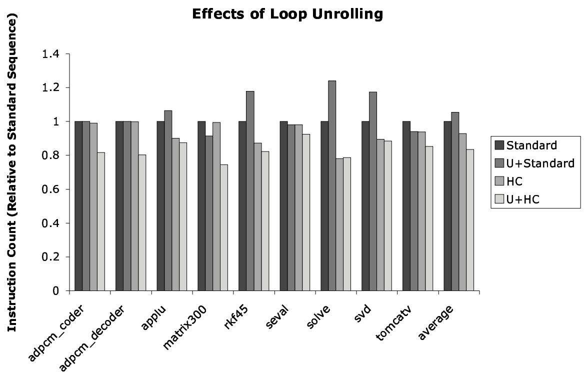


Fig. 1. The effects of adding unroll to a fixed sequence and to the adaptive hill climber. *Standard* refers to the standard fixed sequence and *HC* refers to the hill climber. The *U* modifier indicates the inclusion of unroll for that experiment.

The amount of work required to find a good solution is also critical. We are concerned with search efficiency: performance gain per unit work. Figure 2 shows that the results depend on available opportunity. `adpcm_coder` shows better efficiency with unroll while `solve` shows the opposite result. `rkf45` shows an interesting curve; efficiency with unroll is worse until the search finds a context where unroll pays off; then its efficiency with unroll is better than without it.

Finally, we examined the frequency of peel and unroll in the best sequences from the larger space. Figure 3 displays the average pass frequency for the best 1% of the sequences for each benchmark. Peel is now selected less than half as often as it was before we added unroll. Unroll is selected roughly 11% of the time, showing that the hill climber finds effective uses for it.

3 Conclusion

We improved the performance of code produced by our adaptive compiler up to 25% by addressing a deficiency in its transformation set. Despite the larger search space, the hill climber is usually able to find better compilation sequences with fewer evaluations. These results suggest that the effectiveness of adaptive compilation relies heavily on the capabilities of the underlying optimizations. This kind of evaluation may also lead to a fair basis for comparing transformations.

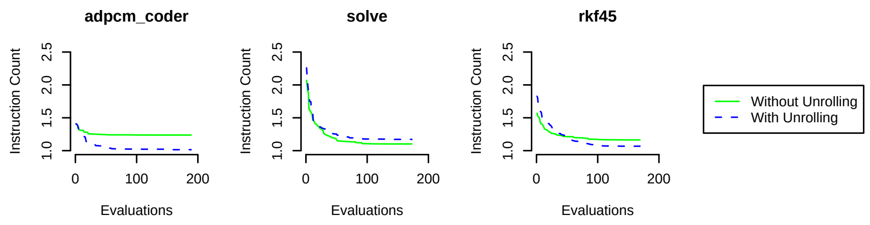


Fig. 2. Search efficiency: average instruction count (normalized against best sequence found) as a function of sequences evaluated

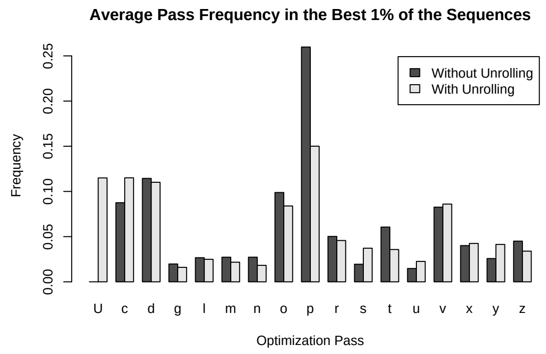


Fig. 3. Average pass frequency in the best 1% of the sequences; *p* is peel and *U* is unroll

References

1. Grosul, A.: Adaptive Ordering of Code Transformations in an Optimizing Compiler. PhD thesis, Rice University (2005)
2. Almagor, L., Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S.W., Subramanian, D., Torczon, L., Waterman, T.: Finding effective compilation sequences. In: LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, New York, NY, USA, ACM Press (2004) 231–239
3. Qasem, A., Kennedy, K., Mellor-Crummey, J.: Automatic tuning of whole applications using direct search and a performance-based transformation system. In: Proceedings of the Los Alamos Computer Science Institute 5th Annual Symposium. (2004)
4. Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M.F.P., Thomson, J., Toussaint, M., Williams, C.K.I.: Using machine learning to focus iterative optimization. In: CGO '06: Proceedings of the International Symposium on Code Generation and Optimization, Washington, DC, USA, IEEE Computer Society (2006) 295–305
5. Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J., Jones, D.: Fast searches for effective optimization phase sequences. In: PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, New York, NY, USA, ACM Press (2004) 171–182
6. Allen, F.E., Cocke, J.: A catalogue of optimizing transformations. In Rustin, R., ed.: Design and Optimization of Compilers. Prentice-Hall, Englewood Cliffs, NJ, USA (1972) 1–30
7. Sreedhar, V.C., Gao, G.R., Lee, Y.F.: Identifying loops using dj graphs. ACM Trans. Program. Lang. Syst. **18**(6) (1996) 649–658
8. Cooper, K.D., Simpson, L.T., Vick, C.A.: Operator strength reduction. ACM Trans. Program. Lang. Syst. **23**(5) (2001) 603–625
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4) (1991) 451–490

An Effective Local Search Algorithm for an Adaptive Compiler

Yi Guo, Devika Subramanian, and Keith D. Cooper

Rice University, Department of Computer Science,
Houston, Texas 77005 USA
{yguo,devika,keith}@rice.edu

Abstract. Most algorithms currently used to find good compilation sequences in an iterative adaptive compiler, such as genetic algorithms and hill climbing, search in the space of sequences of fixed length. In this paper, we argue that restricting the search to fixed-length sequences limits the ability of search algorithms to find good sequences for some benchmarks. We propose a new local search algorithm that uses greedy construction and cleanup to effectively explore the neighborhood of a start sequence by randomized insertion and deletion of transformations. Preliminary experimental results show that the quality of the local minima found by our local search algorithm are superior to those sequences found by GAs and HCs, and are close to the best sequence we know. Such local minima are found with significantly lower search effort than GAs and HCs working with fixed-length sequences.

1 Introduction

Over the last several years, various groups [1, 6, 3] have studied the code transformation selection and ordering problem in an iterative adaptive compiler. Several search techniques for biased random sampling of the combinatorial space of program-specific optimization sequences have been proposed. Genetic algorithms (GAs) and hill climbing are two popular search algorithms implemented in research iterative compilers. While the choice of specific parameters may vary, these algorithms share one common characteristic: solutions are represented as fixed-length sequences of code transformations and the length of the solution is not varied during the search process. This fixed-length framework is dictated by the use of standard genetic operators, i.e. 1-point crossover and mutation, used in GAs to generate variations for the next generation, and the use of Hamming distance to define neighbors in hill climbing.

Figure 1 presents evidence that current search algorithms based on the fixed-length framework do not find the best solutions for some benchmarks. For `spline`, even after 500 trials, our optimized genetic algorithm achieves less than 45% of the performance speedup of the best known sequence. We believe there are two reasons for this: (1) the fundamental GA operations do not explore the space of optimization sequences effectively. The mutation operator, which randomly generates point variations of a sequence, makes local changes very slowly,

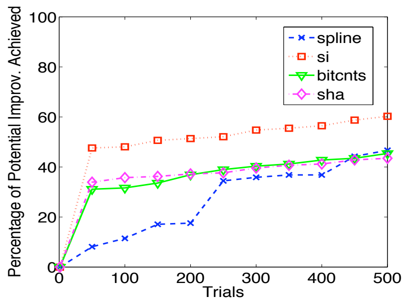


Fig. 1. The percentage of potential improvement achieved by an optimized GA over 500 trials for four benchmarks.

since mutation rates are generally set low. The 1-point crossover operator in GAs is quite destructive and does not generate semantically meaningful variations, and (2) The constraint of fixed-length sequences also limits the range of performance gains that can be achieved. Some benchmarks need fairly short compilation sequences. For those cases, a fixed-length GA spends valuable time and resources learning no-op subsequences to pad the sequence to the full predetermined length. Other benchmarks need longer sequences, and the GA fails to realize the full benefit of the range of optimizations available in the compiler. Attempts to fix these two inherent weaknesses in exploration strategy by tuning algorithm parameters such as the population size, as well as the size of the fixed-length representation, yield little improvement in search performance for several benchmarks.

In this paper, we show that if the neighbor set of a given sequence is explored effectively, the local minima have quality competitive with the best sequences we know. Instead of defining neighbors by Hamming distance, we define neighbors by edit distance, and use greedy construction and cleanup to generate a richer set of meaningful variations by insertion and deletion of transformations. Preliminary experimental results for some benchmarks on the SPARC backend show that the local search algorithm can significantly outperform GAs and hill climbers working in the space of fixed-length sequences.

2 Related Work

Schielke's 1999 paper [2] appears to be the first use of a GA to find compilation sequences. He showed improvement in both code size and execution speed. The framework of the GAs used in the paper is similar to those used in current research iterative compilers. Several techniques have been proposed to improve the GAs' searching performance without changing its fixed-length framework. Kulkarni et al. [6] proposed techniques to speed up searches for compilation sequence in genetic algorithms by detecting and removing redundant trials of equivalent programs, and prohibiting certain dormant or disabled transformations.

Statistical and machine learning techniques have been used to improve the performance of searching. Agakov et al [1] selected a set of benchmarks and learned an offline model for each benchmark. When given a new program, the model of the benchmark that is most similar to the new program is used to focus the search space.

In [3], Grosul describes and compares several variations in an adaptive compiler and found that GAs outperform hill climbing and other algorithms on a budget of several hundreds to a few thousand compilations. In this paper, we use the same experimental setup and compare our local search algorithm to the genetic and hill climbing algorithm in [3]. We show that our local search algorithm significantly outperforms GAs and hill climbing by finding better sequences with far fewer compilations.

Several groups have worked on the problem of finding good parameter settings for specific transformations. Triantafyllis et al. [7] demonstrated the promise of using multiple compilation configurations in a practical compiler. Zhao et al. [8] described an approach for modeling interactions in a predictive framework. Kisuki et al. [5] have used various search algorithms to find good optimization settings for loops in numerical kernels.

3 Neighbor Exploration: Finding Local Minima

The definition of neighbor is fundamental to search algorithms. In the fixed-length framework, it is easy to define neighbors by Hamming distance: two sequences of the same length are considered neighbors if they differ in exactly one character. The experiments in [3] show that the hill climbers using Hamming distance do not deliver satisfactory results. Our local search algorithm defines neighbors by edit distance, i.e., two sequences are considered neighbors if one can be derived from another by inserting or removing one transformation.

Two procedures are used to find local minima in sequence space by exploring the neighborhood of a starting sequence: cleanup and greedy construction. The cleanup procedure removes transformations that are redundant or detrimental to the quality of the given sequence. Such transformations can appear during both greedy construction and random sequence generation. Greedy construction extends the base sequence one transformation at a time. At each step it picks a transformation and inserts it into the position that delivers the most improvement. If a transformation does not yield improvement, it is discarded.

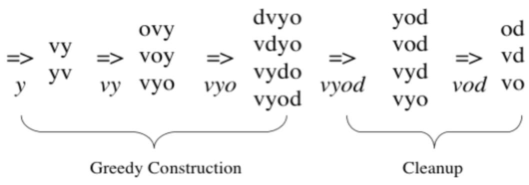


Fig. 2. Neighbor Exploration

Figure 2 shows an example of greedy construction and cleanup. Transformation *v, o* and *d* are inserted into sequence *y*, and transformation *y* is removed by cleanup. Our algorithm starts the search from random sequences. Local minima are found by iteratively running greedy construction and cleanup. Detailed algorithm description can be found in our technical report [4].

4 Experiments

In this paper, we use 16 transformations, which are listed in Table 3.1 on page 17 of [3]. They are low-level code transformation based on ILOC, which is a RISC-like assembly language. Our implementation ensures that each transformation can take any valid ILOC program as input and producesa valid ILOC program as output. This feature allows us to run the compilation transformations in arbitrary order, which is critical for an adaptive compiler. When using the default compilation sequence *rvzcodtvzcod*, the performance of the code generated by our ILOC compiler is comparable to the *GCC* compiler using *-O2* flag.

We compare our local search algorithm to GA and Hill Climbing (HC). The parameter settings for GA and impatient hill climbing(HC-10) are described on page 76 in [3]. For GA, we tried three length settings, 15, 20 and 25, and the curve represents the best among the three. The sequence length for HC is fixed at 15. Figure 3 shows the search performance for three algorithms within 1000 trials. The speedup of a sequence is normalized to 0-100% where the default sequence *rvzcodtvzcod* is set to 0% and the best sequence is set to 100%.

According to Figure 3, our search algorithm excels other algorithms after 200 trials, and after 1000 trials, the quality of sequence we found is close to the best. Table 4 shows the length of the best sequence we known and the best GA's length setting. The best length settings of GA is program-specific, and there is no obvious relation to the length of the best sequence.

Benchmark	Source Suite	Len. of the Best Seq.	Best GA's Len. Settings
spline	fmm	13	20
si	spec	24	20
bitcnts	mibench	29	25
sha	mibench	29	20

Table 1. Benchmark

5 Conclusion

This paper considered two hypotheses for the poor performance of GAs and HCs on complex compilation sequence spaces for some benchmarks. The first is the fixed-sequence length limitation and the second is the choice of genetic operators for constructing variations to explore during search. We introduces a local search algorithm with a richer neighborhood definition generating variable length sequences. We demonstrate that this new local search algorithm outperforms GAs and HCs on a set of benchmarks, both on the quality of solutions and the search effort needed to find them.

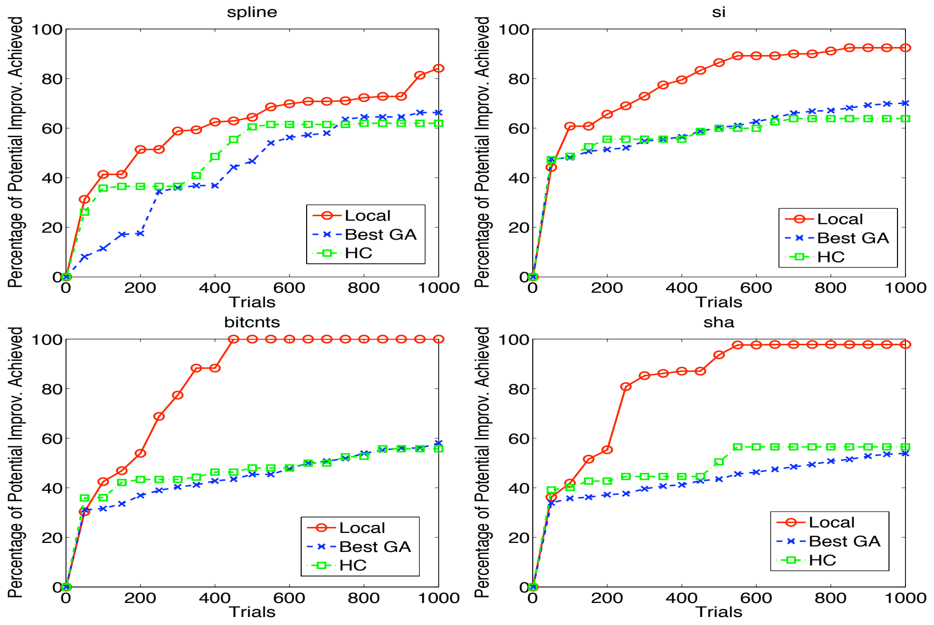


Fig. 3. Average percentage of potential improvement achieved by our local search algorithm, GA and HC on the SPARC backend for four benchmarks

References

1. Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M. F. P., Thomson, J., Toussaint, M., and Williams, C. K. I. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization* (2006).
2. Cooper, K. D., Schielke, P. J., and Subramanian, D. Optimizing for reduced code space using genetic algorithms. In *LCTES '99* (1999).
3. Grosul, A. *Adaptive Ordering of Code Transformations in an Optimizing Compiler*. PhD thesis, Rice University, 2005.
4. Guo, Y., Subramanian, D., and Cooper, K. A new local search algorithm for effective exploration of compilation sequences. Tech. rep., Rice University, 2006.
5. Kisuki, T., Knijnenburg, P. M. W., and O'Boyle, M. F. P. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques* (2000).
6. Kulkarni, P. A., Hines, S. R., Whalley, D. B., Hiser, J. D., Davidson, J. W., and Jones, D. L. Fast and efficient searches for effective optimization-phase sequences. *ACM Trans. Archit. Code Optim. 2*, 2 (2005), 165–198.
7. Triantafyllis, S., Vachharajani, M., Vachharajani, N., and August, D. I. Compiler optimization-space exploration. In *CGO '03: Proceedings of the international symposium on Code generation and optimization* (2003).
8. Zhao, M., Childers, B., and Soffa, M. L. Predicting the impact of optimizations for embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems* (2003).

Basic-block Reordering Using Neural Networks

Xianhua Liu, Jiyu Zhang, Kun Liang, Yang Yang and Xu Cheng

Microprocessor Research and Development Center,
Peking University, Beijing, P.R.China, 100871
{liuxianhua, zhangjiyu, liangkun, yangyang, chengxu}@mprc.pku.edu.cn

Abstract. Basic-block reordering is a compiler optimization technique which has the effect of reducing branch cost and I-Cache misses by rearranging code layout. In this paper, we present our basic-block reordering method which detects typical structures in the control-flow graph. It uses the architecture-specific branch cost model and execution possibilities of control-flow edges to estimate the possible layout costs of specific sub-structures. The layout with the minimal cost estimation would be chosen. We further investigate a new approach to use neural network to predict execution possibility for each edge. We choose a set of programs and record particular static information of the edges in the typical structures. These data includes the knowledge about the relationship between static program features and dynamic behavior, and is fed to train the neural network. In this paper, we adopted an improved back propagation neural network. The algorithm has been implemented based on a 5-stage pipeline UniCore architecture. The experiments show that it improves programs' performance well, and execution possibility of edges may be predicted using machine learning techniques.

Keywords: neural network, basic-block reordering, program structure, profile guided optimization

1 Introduction

The control-flow information is usually represented by control-flow graph during compiler optimization. The binary or assembly is finally laid as one-dimension sequence in the memory, which brings a gap between two-dimension graph-represented program and sequential one. Compilers will arrange the layout of binaries at the end of the compilation process. In modern pipelined microprocessors with hierarchical memory system, the performance of a binary program will be affected by its code layout in most cases. Whether the branch will be taken and the branch target address cannot be known until the branch reaches a late pipeline stage. This often brings a performance loss, which is more obvious in deeper pipelines. For example, in Digital Alpha 21164, the target address is available at the sixth stage and the average performance loss per branch prediction miss is 11 cycles. Modern processors have introduced various branch prediction strategies to improve branch performance. If the layout of one binary matches the processor's branch prediction strategy, the performance might be further improved. The layout of a binary may also affect instruction cache performance. The target of a branch might be laid un-continuously with itself. When a branch is taken, if the branch instruction and its target are not in the same cache line, there may be some un-useful instructions fetched into the cache lines but never executed. This leads to instruction cache efficiency decrease. Further more, if frequently-interacted code fragments are laid close to each other, the possibility of conflict in instruction cache will be reduced.

Our algorithm analyzes the structures of the program and gives the optimal layout for each local sub-structure [23]. We combine structural analysis and machine learning techniques to guide the layout process. While achieving good experimental results, most of basic reordering methods are implemented based on profiling information of the program execution. One of the drawbacks of profile-based methods is the additional work of profile generation and selection. We attempted to find out if the profile information of one execution trace can be used to predict another execution trace of the program or the behavior of another program. Our method has two main phases. In the first phase the analyzer identifies various local structures from a set of programs and collects relevant profiling information. Then we use that to train a neural network mapping static features associated with each control-flow edge to an execution possibility prediction. In the second phase, we apply the trained neural network to predict the execution possibility for each control-flow edge, to feed our basic-block reordering algorithm based on local structural analysis.

Basic-block reordering using machine learning methods has several advantages. First, compared with profile-based methods, it can save programmers' work of producing and choosing proper profile information to guide compiler optimization. Second, traditional heuristic based methods require compiler writers to analyze many static features, so that to find out which set of features affects the execution of the programs most and how. Machine learning methods automatically select the information.

The rest of this paper is organized as follows. In Section 2, we discuss some related works. In Section 3, we introduce our architectural branch cost model and basic-block layout cost model. We also describe the structural analysis based basic-block reordering algorithm. Our neural network used to learn the mapping static program features to execution possibilities is addressed is Section 4. Section 5 shows the experimental results and the conclusion is given in Section 6.

2 Related Works

There are a number of techniques reducing branch costs, both hardware-based and software-based. Most of them use past program behavior to predict its future actions. One of the most popular compiler optimization techniques is suggested in [1]. In that paper, Pattis and Hansen use profile information of execution frequency of each edge to arrange the code placement. Although it is a greedy algorithm and the optimal layout cannot be guaranteed, many compilers and basic-block reordering techniques are based on it [2][4][5][6][12][15]. The idea of taking the different branch costs of the specific architecture into consideration is described in [12]. In [2], the authors reduce a limited form of the reordering problem to the Directed Traveling Salesman Problem which is NP-hard. They also present experimental results using heuristic algorithms from training and testing on different data sets, which shows only a little reduction of the benefits from the code placement algorithms.

Compiler writers have crafted many heuristics over the years to approximately solve NP-hard problems efficiently [26]. Profile-based optimizations require effective profiles which are not deeply discussed in most of the works. The experimental results in [2] shows a possibility of using past program behavior to predict its future actions, like many hardware-based branch prediction techniques do. Meanwhile, the authors in [24] investigate an approach to uses a body of existing programs to predict the branch behavior in a new program at compile-time. In that paper, the authors use machine learning techniques and show their efficiency in predicting the branch behavior. Inding a heuristic that performs well on a broad range of applications is a tedious and difficult process. J.Cavazos induces heuristics in instruction scheduling [27], it also shows that inexpensive and static features can be successfully used in scheduling.

In this paper, we investigate ways to combine our structural analysis based method and the machine learning techniques to guide the layout process, and intend to get some hints related to program structures and behaviors.

3 Basic Block Reordering Using Structural Analysis

In this section, we give a briefly description to our local structural analysis based basic-block reordering method. A more detailed one can be found in [23]. As mentioned before, greedy algorithms usually cannot get the optimal layout. Modern programs are well structured to be divided into typical structures usually. Our method first identifies all local structures of a program. For each structure, we use structure cost model and the execution frequencies of control-flow graph edges to calculate the cost of each possible layout and the one has the minimal cost is chosen. Experimental results show that our method can improve performance by 7% on average which is better than the common used greedy method [23].

3.1 Architecture-Specific Branch Cost Model

In our model, branch cost is defined as the cycles needed to execute the branch/jump instruction (if needed) plus the cost caused by the branch or jump. We are mainly concerned with reducing branch/jump cost, and instruction cache performance may be improved as well. In this paper, we will call branch instruction or jump instruction all as branch for convenience.

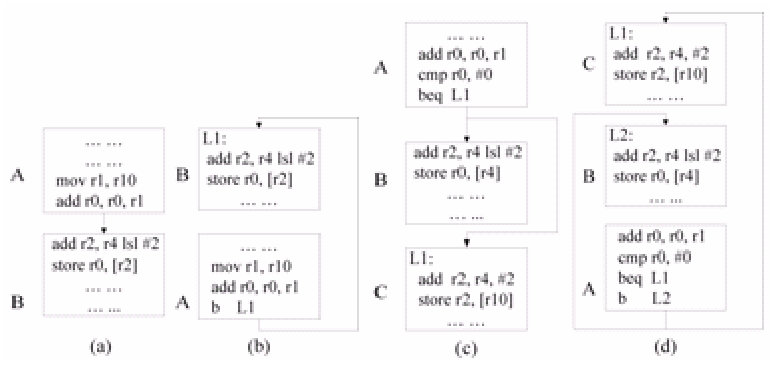
For a typical single-issued pipelined processor with branch prediction mechanism, the branch cost can be calculated as follows: *Branch cost = Cost of branch × Numbers of branches + Cost of branch prediction miss × Numbers of branch prediction misses.* Here the number of branches stands for the dynamic number of branches executed.

Different processors may have different branch cost models due to architectural difference. We analyze and conclude the branch cost model for UniCore-I processors.

But the method isn’t limited to them and can be used on many other pipelined processors.

UniCore-I is a single-issued RISC microprocessor with five pipeline stages. Its branch prediction mechanism is to assume the fall-through path is always executed [14]. Since whether a branch will be taken is not determined yet, it continues to fetch the following instructions in the fall-through path. If the branch is taken, these instructions in pipeline will be squashed, and re-fetch right instructions. This branch prediction strategy is very common to be seen in HP PA-RISC, Alpha AXP-21064 and many other processors. In UniCore-I processor, a branch instruction itself will take 1 cycle in the pipeline. If the branch is taken, it will cost extra 2 cycles till branch target is calculated. Thus, a taken conditional branch will take 3 cycles in all and a not-taken conditional branch takes one. An unconditional jump will always take 3 cycles.

The branch cost is directly related to the edges in the control-flow graph. For a control-flow graph G, the branch cost of an edge e=head→tail is defined as the cycles needed to go from head to tail, and is represented as *Cost(e)*. Given an architecture, *Cost(e)* is related to the characteristics of basic-block Head and tail and their relative positions in the linear space. Figure 1 shows the possible situations:



```
(a)
A:  ... ...
    ... ...
    mov r1, r10
    add r0, r0, r1
B:  add r2, r4 lsl #2
    store r0, [r2]
    ... ...
    ... ...

(b)
B:  L1:
    add r2, r4 lsl #2
    store r0, [r2]
    ... ...
A:  ... ...
    mov r1, r10
    add r0, r0, r1
    b   L1

(c)
A:  ... ...
    add r0, r0, r1
    cmp r0, #0
    beq L1
B:  add r2, r4 lsl #2
    store r0, [r4]
    ... ...
    ... ...
C:  L1:
    add r2, r4, #2
    store r2, [r10]
    ... ...

(d)
C:  L1:
    add r2, r4, #2
    store r2, [r10]
    ... ...
B:  L2:
    add r2, r4 lsl #2
    store r0, [r4]
    ... ...
A:  add r0, r0, r1
    cmp r0, #0
    beq L1
    b   L2
```

Fig. 1. Different Branch Cost of Different Edges in Control-Flow Graph

In Figure 1-(a), basic-block A has only one successor and can go through to B directly with no branch instructions, so the cost of edge A→B is 0; In Figure 1-(b), A has only one successor B, but B is not laid just following A, thus there need be one jump instruction in the end of A, thus edge A→B costs 3 cycles; In Figure 1-(c), A has two successors with B just following it, thus there should be one conditional branch in the end of A and edge A→B costs 1 cycle while edge A→C costs 3 cycles; In Figure 1-(d), A also has two successors B and C, but neither of them is laid just following A, so there should be two branches at the end of A and edge A→B costs 3 cycles while edge A→C costs 4 cycles.

In UniCore-I, for a control-flow edge e=A→B, let *Profit(e)* be the profit of laying B directly following A. It is the cost of not laying B right following A minus the cost of laying B right following A, as shown in Table 1.

Table 1. The *Profit(A→B)* in UniCore-I

	Cost of not laying B directly following A	Cost of laying B directly following A	*Profit(A→B)*
A has only one successor	3 cycles	0	3 cycles
A has more than one successors	3 or 4 cycles	1 cycle	2 or 3cycles

The case that *Cost(e)*=4 is only when none of A's successors is laid directly following A, which cannot be determined before the layout is finished, and is not very common. So we assume the *Profit(A→B)* to be 2 when A has more than one successors. Thus in our model, according to the number of A's successors, *Profit(A→B)* is calculated as follows:

$$Profit(e) = \begin{cases} 3, & \text{A has 1 } successor \\ 2, & Other \end{cases}$$

The final profit-weight of the edge *Key(e)* is calculated as follow: *Key(e)=Profit(e) ×Frequency(e).*

3.2 Local Structural analysis and Optimization

PH algorithm use profile information of execution count of each edge to arrange the code placement. Pattis and Hansen describe two code placement algorithms. The bottom-up one begins with the edge with the highest execution frequency and tries to arrange the tail node of the edge immediately following the head node. It's a greedy algorithm which cannot get the optimal layout in some cases. See Figure 2 as an example. Basic-block A, B and C construct an if-then structure, execution frequency of each edge is marked beside. Since *Freq(A→C) > Freq(A→B) and Profit(A→C) = Profit(A→B)* = 2, *Key(A→C) > Key(A→B)*. Thus the program layout will be A-C in PH algorithm, not including block B in the chain. The total branch cost will be 6500 + 3500*3 + 3500*3 = 27500 (cycles). But if we lay them as A-B-C, the total branch cost will be 3500 + 6500*3 = 23000 (cycles). This would be better than the former.

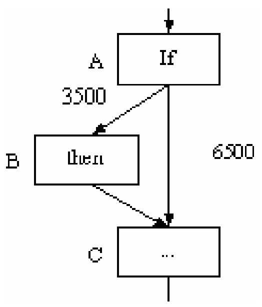


Fig. 2. Example of If-then Structure

Modern programs always have good structures and their control-flow graph can be divided into typical structures. Most of these structures can be represented as nine types [9]. Among these structures, the improper interval schema doesn't always appear and has no fixed structure. The self-loop structure has only one basic-block, while case/switch structure are represented as jump tables in machine codes, so these structures aren't concerned in our work. We define 7 typical structures and analyze their optimal layout given the execution frequencies of all edges, which are shown in Figure 3.

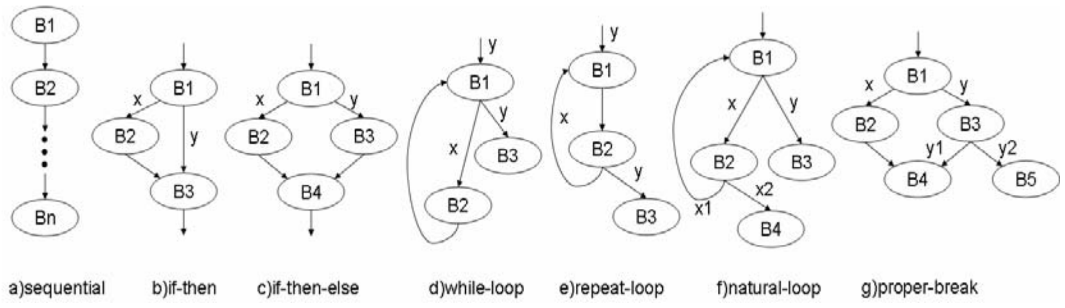


Fig. 3. Seven Typical Structures in Control-flow Graph[1]

We will take if-then structure as an example to introduce our algorithm getting the optimal local layout. *Freq(e)* represents the execution count of the edge e in profile. As shown in Figure 3-(b), in if-then structure, B1 has two successors, thus the possible layout includes: B1-B2-B3 or B1-B3 (B2 is set to be a separate node). For the former layout, the branch cost is:

$$\begin{aligned} Cost_{B1\text{-}B2\text{-}B3} &= Freq(B1\text{->}B2) * Cost(B1\text{->}B2) + Freq(B2\text{->}B3) * Cost(B2\text{->}B3) \\ &+ Freq(B1\text{->}B3) * Cost(B1\text{->}B3) \\ &= x+3y \end{aligned}$$

Similarly, for the latter layout, the branch cost is $Cost_{B1\text{-}B3}=6x+y$. So if B1-B2-B3 is better, it demands $Cost_{B1\text{-}B2\text{-}B3} \leq Cost_{B1\text{-}B3}$, that is $y \leq 2.5x$. Similarly, if the algorithm chooses B1-B3, then it demands $y \geq 2.5x$. If $y=2.5x$, the first layout is chosen for better instruction cache locality. The item of if-then in Table 2 shows the above results. Other structures' optimal local layouts under different frequencies of edges are also listed in Table 2.

Table 2. Optimal Local Layout for Each Structure under Different Conditions

Structure	Optimal Local Layout	Condition
Sequential	Sequential	N.A.
If-then	B1-B2-B3	y ≤ 2.5x
	B1-B3	y >2.5x
If-then-else	B1-B2-B4	y ≤ x
	B1-B3-B4	y > x

[1] The weight beside each edge represents the execution counts of the edge according to the profile information

While-loop	The other predecessor of B1's is laid directly before B1	B2-B1-B3	$y \leq x$
		B1-B3	$y > x$
	Other	B2-B1-B3	N.A.
Repeat-loop	B1-B2-B3		N.A.
Natural-loop	The other predecessor of B1 is laid directly before B1	B1-B3, B2-B4	$y > x$
		B1-B2-B4	$y \leq x$
	Other	B1-B3, B2-B4	$y > x$ and $x_1 < x_2$
		B1-B2-B4	$y \leq x$ and $y < 2x_2$
		B2-B1-B3	Other
Proper-break	B1-B2-B4, B3-B5		$x > y$
	B1-B3-B4		$x \leq y$ and $3x+2y_2<2y_1$
	B1-B3-B5,B2-B4		Other

Thus we can get the table of optimal local layout for each structure under different conditions. In this phase of the algorithm, local structural analysis and layout optimization is performed: The algorithm traverses the whole graph and searches for typical structures. Once a typical structure is identified, the algorithm will reorder the basic-blocks of it according to the above table and the particular execution frequencies of the edges. Let's take the weighted control-flow graph in Figure 2 for example. In that case, *Freq(A->B)=6500<2.5×3500=2.5×Freq(A->C)*, so the optimized layout is A-B-C.

We describe our algorithm in pseudo code as following:

```
procedure reorder (N, E)
 begin
sort_edge(E); //Sort all edges by Profit(e)
for e in E do //visit all edges by order of Profit(e) and link them to chains
begin
   if A.visit = false and B.visit = false; //A、B are both not visited
      B.visit := true; A.visit := true;
      A.next := B;
      B.isTail := true; A.isHead := true;
   else if B.visit = false and A.isTail = true // B is not visited, A is tail of a chain
      B.visit := true; A.next := B;
      A.isTail := false; B.isTail := true; // B becomes tail
   else if A.visit = false and B.isHead = true // A is not visited, ,B is head of a chain
      A.visit := true; A.next := B;
      A.isHead := true; B.isHead := false; // A becomes its head
   else if A.isTail = true and B.isHead = true; // A、B are both visited
      A.next := B;
      A.isTail := false; B.isHead := false; //form these 2 chains
end
structural_analysis(N,E); //scan all chains, optimize according table 2 via
structural analysis
connect_chains(); //link all chains to form the binary
 end
```

```
procedure structural_analysis (N,E)
 begin
for B in N（Deep-First-Order）do
begin
   // if-then structure?
   if Succ(B)={ $V_1$ ,$V_2$}
      if Pred($V_1$)={B} &&Pred($V_2$)={B} &&Succ($V_1$)={$V_2$} &&Profit (B→$V_1$) + Profit($V_1$→$V_2$)>Profit (B→$V_2$)
         B.next := $V_1$; $V_1$.next := $V_2$; $V_1$.visit := true;
      if Pred($V_1$)={B} &&Pred($V_2$)={B} &&Succ($V_2$)={$V_1$} &&Profit (B→$V_2$) + Profit($V_2$→$V_1$)>Profit (B→$V_1$)
         B.next := $V_2$; $V_2$.next := $V_1$; $V_2$.visit := true;

   //test whether basic-block B and its succeeds is a typical structure, select optimal
   layout according to Table2
   ……

end
end
```

4. Predictions with Artificial Neural Network

In this section, we use artificial neural network to predict the execution probability of edges in typical structures given above. Our idea is generally described as follows. A set of programs are gathered and particular static information about edges in the typical structures is recorded. Relevant dynamic behavior is associated with each edge while profiling. We have accumulated a dataset about the relationship between static program features and dynamic behaviors. This dataset is used to train the neural network to predict the behaviors of edges. In the dataset, the edges with similar static features may exist in programs not in the training set or the programs with different inputs.

This section contains two main parts. First, we introduce the static feature set extracted for the training process. Second, we describe the artificial neural network used in our prediction.

4.1 Static Feature Set for Prediction

To apply the neural network to our problem, the static feature set as input of neural network is firstly determined. We record the static features for each edge that are used in the algorithm above in each typical structure (see Table 3). Some features are used to identify typical structures and the edges constructing them, others are some properties related to the edges.

Table 3. Selected Static Features for Edges

No.	Feature name	Feature description
1	SS_No.	Unique number to identify typical structure.
2	Edge_No.	Unique number to identify edges.
3	I_last_of_head	The operation code of the last instruction in the basic-block of edge's head.
4	Br_direction	Branch direction.
5	I_pre_last	The operation code of the instruction before the last instruction in the edge's head
6	Is_bl	Whether the last instruction in the edge's head is a
7	Is_swi	Whether the last instruction in the edge's head is SWI.

B. Calder [24] brings neural network and static features of program in their evidence-based static branch prediction (ESP). We've referred these suggestions to choose these features. The most important difference is that, [24] considered every single branch in a program, while we are concerning edges in an identified typical structure. A lot of static information, such as whether the edge is a part of a loop, etc, has already been contained in the type of a typical structure.

We choose the feature set in Table 3 based on several criteria. First, we adopt information which would be likely predictive of behavior. This information includes two parts. One is a unique number for each typical structure. The other unique number is for edges. Second, we use encode some information related to a given edge according to classical static branch prediction methods, such as the operation code of the last instruction in the edge's head basic-block, branch direction and etc.

We profile the programs to collect information on execution counts of edges and its head basic-blocks. Since the execution counts of edge varies with programs, we normalized it by its head basic-block execution counts, that is, execution probability of edge is execution_counts_of_the_edge / head_basic-block_execution_counts. Finally, we associate the static features of each edge with the corresponding execution probability.

4.2 Artificial Neural Network Building

Our goal is to build a tool that can effectively predict the edge's execution probability based on static program features. It should accurately predict not only for the programs it studied, but also for other programs it is never acquainted with.

We use an improved back propagation neural network, which is not fully connected and has shortcut connections [20]. A cascade model is used to train it. The cascade training model differs from the ordinary ones in the sense that it starts with an empty neural network and then adds neurons incrementally. The basic idea of this model is that some neurons are trained separately, and the most promising candidate is inserted. Then the output connections are trained and new candidate neurons is prepared. The candidate neurons are created as shortcut connected neurons in a new hidden layer, which means that the final network will consist of a number of hidden layers with one connect neuron in each. We choose RPROP algorithm as its internal training algorithm of this model [20]. And the candidate neurons' activation functions may be one of the followings.

The neural network we adopted has only one hidden layer. Thus, the structure is 7-X-1. Here X is the number of neurons in the hidden layer, which is determined in the learning phase. The activation function for the candidate neuron may be one of the following:

Sigmoid: $y = \frac{1}{1+e^{-2\times s\times x}}$;

Sigmoid_symmetric: $y = \tanh(s \times x)$;

Gaussian: $y = e^{-x\times s\times x\times s}$;

Gaussian_symmetric: $y = e^{-x\times s\times x\times s} \times 2 - 1$;

Elliot: $y = \frac{\frac{s \times x}{2}}{1 + |s \times x|} + 0.5$;

Elliot_symmetric: $y = \frac{s \times x}{1 + |s \times x|}$;

The output function is,

$$y = \begin{cases} \tanh(s \times x) & \text{if } \tanh(s \times x) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Here x is the biased linear combination of the outputs of the hidden candidate neurons. The learning criterion is to minimizing the Mean Square Error. More details and discussions about the neural network are available in the next section.

5. Experimental Results and Analysis

In this section we present our experimental results. We have implemented the method described above as a post pass of GNU toolchain (gcc 3.2.1). We use integer benchmarks in SPEC2000 as our neural network training data and use several programs from MediaBench[14] as our final benchmark test data. Each program is compiled with GCC "–O2" level of optimization. The performances are measured using sim-pipeline: a cycle-accurate five-stage pipeline UniCore simulator modified from sim-outorder in simplescalar[16]. The main configuration for UniCore-I processor simulator is listed in Table 4, and the benchmark programs are briefly described in Table 5.

Table 4. Simulation Parameters of UniCore-I Processor

Branch Prediction Scheme	Not taken
Level 1 I-Cache	

Capacity	8K
Associativity	2
Block Size	32 Bytes
Replace Scheme	Round-Robin
Level 1 D-Cache	
Capacity	8K
Associativity	4
Block Size	32 Bytes
Other Schemes	Round-Robin, Write back, Write allocate
Memory Access Latency	
First Chunk	20 Cycles
Inter Chunk	2 Cycles

Table 5. Benchmark Description

Name	**Description**
adpcm	Adaptive differential pulse code modulation, one of the simplest and oldest forms of audio coding (encode / decode)
epic	An experimental image compression utility (epic / unepic)
pegwit	Public key encryption and authentication (encode / decode)
jpeg	Standardized compression method for full-color and gray-scale images (encode / decode)
mesa	A 3-D graphics library clone of OpenGL (mipmap / osdemo / texgen)
mpeg2	A standard for high quality digital video transmission (encode / decode)

Experimental Results

Experimental results are shown in Figure 4-6 (ORI means original program. PROFILE means using profiling optimization directly. ANN means using artificial neural network built above).

While applying the cost model mentioned in Section 3 on UniCore-I architecture, the branch_inst_cost is 1 cycle and branch_miss_cost is 2 cycles, so that the branch cost of UniCore-I is calculated as *branch instruction counts + 2* branch prediction misses*. As shown in Figure 4, ANN can get an average branch cost reduction of 25%, which is surprisingly close results compared with PROFILE (28%).

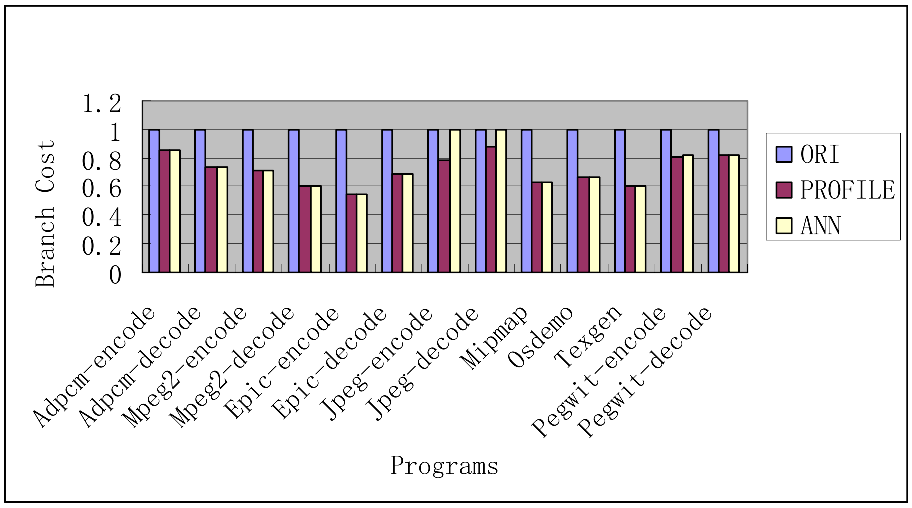


Fig. 4. Normalized Branch Cost

The influence of instruction cache misses is compared and shown in Figure 5. Generally speaking, both the PROFILE and ANN methods achieve a considerable improvement on I-Cache efficiency. However, sometimes ANN wins PROFILE. One possible reason might be that our basic-block reordering algorithm is not specially tuned for I-Cache and the optimal layout is only inside single structures. ANN brings some random factor into the algorithm and gets good result sometimes.

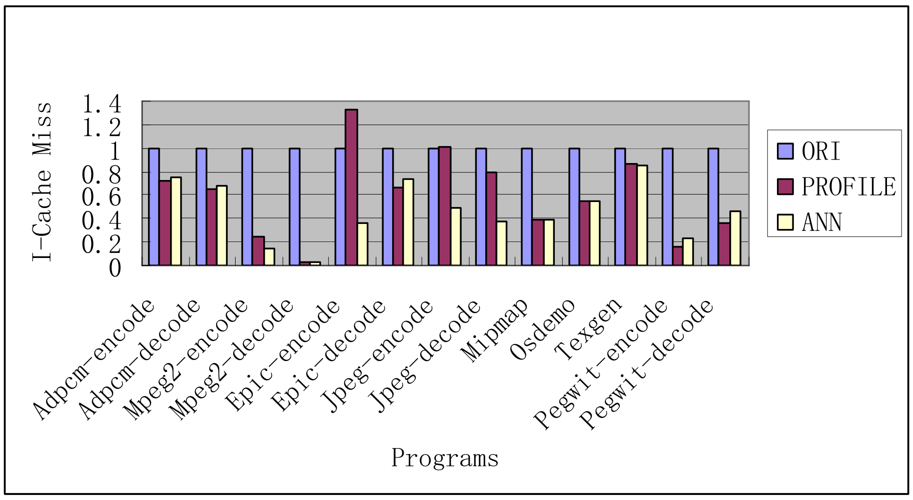


Fig. 5. Normalized Instruction Cache Misses

The comparisons on the whole performance is shown in Figure 6. On this aspect, PROFILE and ANN make similar improvement, 9% and 8% separately.

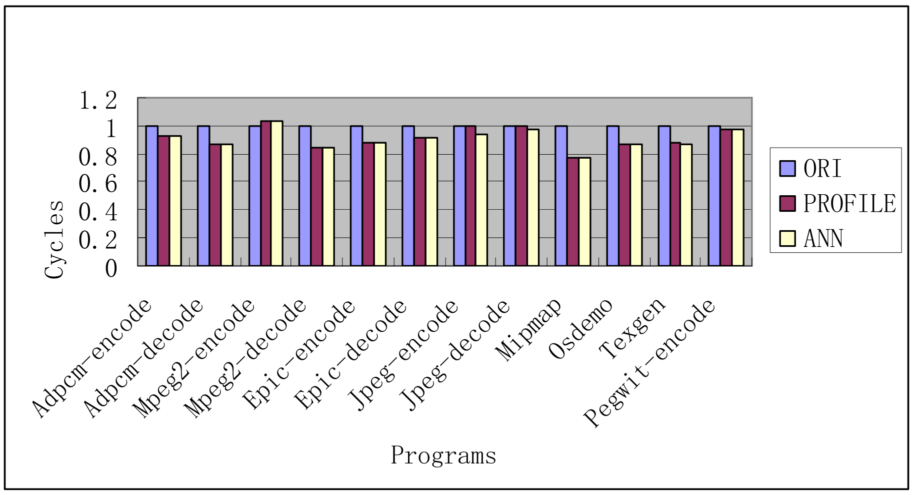


Fig. 6. Normalized Cycle Counts

As we can see above, the result of using artificial neural network to predict edge execution probability for the algorithm is sometimes even better than using profile information and sometimes worse. And on average, they are very close. The result is better than our primal assumption, especially in the situation that the training of the neural network uses different set of programs from the testing benchmarks. Future analysis is still needed in order to get further understanding.

6. Conclusions and Future Work

In this paper, we described our basic-block reordering method which combines program structural analysis and the neural network technique to give a proper layout for each typical structure. Experiments show that this method can improve program performance and structural analysis based feature extraction can be effective on static prediction of branch possibility. Program structural analysis can provide detailed information about the control flow structures, thus it may also provide a set of important and effective static features for other optimizations based on machine learning. Computer architecture and compiler optimization are related to many concrete applied fields. As the computer systems are becoming more and more complex, it is also harder for researchers to derive conclusions from so many factors and details. We believe applying machine learning and other techniques which are commonly used in computer application fields upon the researches on computer system itself can further help researchers understand and optimize the system.

References

1 K.Pettis, R.C.Hansen, Profile Guided Code Positioning, Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, Vol. 25, No. 6, June 1990, pp. 16-27.
2 S.McFarling, Program optimization for instruction caches. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, April 1989.
3 W.Hwu and P.Chang, Achieving High Instruction Cache Performance with an Optimizing Compiler. In 16th Annual International Symposium of Computer Aritecture, 1989, pp.242-251.
4 D. Zhu, Research on Branch Mechanism for UniCore Architecture. Ph.D. thesis, 2004.
5 Y. Gao, Basic Block Reordering on UniCore Architecture, master thesis, 2005.
6 R.Muth, S.K.Debray, S.A.Watterson, and K.De Bosschere, alto: a link-time optimizer for the Compaq alpha. Software – Practice and Experience, 31(1):67-101, 2001.
7 C.Young, D.S.Johnson, D.R.Karger, and M.D.Smith, Near-Optimal Intraprocedural Branch Alignment, Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation, June 1997, pp. 183-193.
8 M. Sharir, Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers, Computer Languages, vol. 5, nos 3/4, 1980.
9 S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann, 1997.
10 UniCore32 ISA and Programming Manual. Microprocessor Research and Development Center of Peking University, 2002.
11 B.D.Bus, B.D.Sutter, L.V.Put,D.Chanet, and K.D.Bosschere, Link-Time Optimization of ARM Binaries. Proceedings of the 2004 Conference on Languages, Compilers, and Tools for. Embedded Systems, June 2004, pp. 211–220.
12 B.Calder and D.Grunwald, Reducing Branch Costs via Branch Alignment. Proceedings of the 6th International Conference on Architecture Support for Programming Languages and Operating System. Oct. 1994, pp. 242-251.
13 N.Gloy, T.Blackwell, M.D.Smith and B.Calder, Procedure placement using temporal ordering information. Proceedings of the 30th Annual International Symposium of Microarchitecture, Dec.1997, pp.303-313.
14 C.Lee, M.Potkonjak, W.H.Mangione-Smith, MediaBench: a tool for evaluating and synthesizing multimedia and communication systems. Proceedings of the 30th Annual International Symposium on Microarchitecture, Dec.1997.
15 R.Cohn, P.Goodwin, G.Lowney and N.Rubin, Spike: an optimizer for Alpha/NT executables. USENIX Windows NT Workshop, August 1997.
16 http://www.simplescalar.com/
17 D.A.Jimenez, Code Placement for Improving Dynamic Branch Prediction Accuracy, Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI), pp. 107–116, June, 2005.
18 H.Aydin and D.Kaeli, Using cache line coloring to. perform aggressive procedure inlining. ACM Computer Architecture News, vol. 28, no. 1, 2000.
19 A.H.Hashemi, D.R.Kaeli, and Brad Calder. Procedure Mapping Using Static Call Graph Estimation. Workshop on the Interaction between Compilers and Computer Architectures, February 1997.
20 M. Riedmiller and H. Braun. A direct adaptative method for faster backpropagation

learning:the RPROP algorithm. Proceedings of International Neural Networks. San Francisco,1993.

21 A.H.Hashemi, D.Kaeli and B.Calder,. Efficient Procedure Mapping using Cache Line Coloring. Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pp. 171-182, June 1997.

22 A.Ramirez, J.L.Larriba-Pey and Mateo Valero, Software Trace Cache, IEEE Transactions on Conputers, Vol. 54, No.1, pp. 22-35, Jan. 2005.

23 X. Liu, Y. Yang, J. Zhang and X. Cheng. A Basic Block Reordering Algorithm Based On Structural analysis. Technical Report. 2006.

24 B. Calder, D. Brunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer and B. Zorn. Evidence-based Static Branch Prediction Using Machine Learning. ACM Transactions on Programming Languages and Systems, 1997.

25 S. Haykin. Neural Networks: A Comprehensive Foundation. Second Edition. Prentice-Hall, Inc,1999.

26 M.Stephenson, S.Amarasinghe, M.Martin, U.M.O'Reilly, Meta Optimization: Improving Compiler Heuristics with Machine Learning. Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI), June, 2003.

27 J.Cavazos, J. Eliot B. Moss, Inducing Heuristics To Decide Whether To Schedule. Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI), June, 2004.

Black Box Approach for Selecting Optimization Options Using Budget-Limited Genetic Algorithms

Guy Bashkansky and Yaakov Yaari

{guy, yaari}@il.ibm.com
IBM Haifa Research Lab

Abstract. Modern compilers present a large number of optimization options covering the many alternatives to achieving high performance for different kinds of applications and workloads. Selecting the optimal set of optimization options for a given application and workload becomes a real issue since optimization options do not necessarily improve performance when combined with other options. The ESTO framework described here searches the option set space using various types of genetic algorithms, ultimately determining the option set that maximizes the performance of the given application and workload. ESTO regards the compiler as a black box, specified by its external-visible optimization options. For the IBM XLC compiler, with some 60 optimization options, we achieved +13% gain over an aggresive base, using 60 iterations on average. We studied a number of search policies given a fixed iterations budget, and showed that exponentially decreasing the width of the search beam gives the best results.

1 Introduction

In modern compilers and optimizers, the set of possible optimizations is usually very large. In general, for a given application and workload, optimization options do not accrue toward ultimate performance. To avoid selection complexity, users tend to use standard combination options, such as -O, which provide stable, high performance for the "average" application. In general, however, these standard options are not the optimal set for a specific application executing its representative workload.

Genetic algorithms (GA) [17, 3] present an attractive solution to this problem of selecting an optimal set of options. The problem is easily mapped to the original problem of gene optimization. The extended time required to reach to a preferred solution is justified by the much longer life of the optimized program. Past related works (e.g. [12], [15]) dealt with tuning specific compilation heuristics. The Expert System for Tuning Optimization (ESTO) was developed to study this GA solution to the general compiler options optimization. The program first computes an initial result using the best-known optimization set, e.g., -O3, and forms an initial generation; randomly, or using some initial knowledge.

Then, at each iteration, the group of organisms (i.e., option sets) that comprise the generation is evaluated on the input workload. The results are sorted and pass through a breeding and mutation stage to form the next generation. This process continues until a termination condition is reached, where the generation results show some kind of stability (by themselves, and/or with respect to previous generations).

Since the option optimization problem exists regardless of the compiler, operating system, or application, we designed ESTO to be fully configurable along all these axes, so that a given compiler can be approached as a black box using solely its user-visible optimization options. Using this reconfigurability, we held studies on two compilers, GCC, and XLC, as well as on the post-link optimizer FDPR-Pro [10]. Each of these optimizers has between 40 to 60 optimization options, some of which have additional parameters. Measuring the SPEC2000/INT suite, we achieved an average gain of +22% over -O1 for GCC, +13% over -O3 for XLC, and +6% over -O3 for FDPR-Pro. The configurability also allows testing various search policies and parallel execution modes. These are described in the body of the paper.

The paper's main contribution is in the application of GA for the problem of selecting an optimal option set for a specific application and workload. Secondly, an adaptive GA is proposed to improve search potential in multi-optima spaces. Finally, we propose a number of decreasing search beam policies to meet a fixed iterations budget.

The paper is organized as follows. Section 2 discusses related work. Section 3 is the core of the paper, presenting the main problem of option selection, and the details of ESTO's genetic algorithm with its different policies. Section 4 explains the configurability aspects of the framework. Experimental results are examined in Sect. 5, and Sect. 6 concludes the paper.

2 Related work

The general approach for tuning compiler optimization for a given application and workload, is by iterative compilation, measuring the performance and using it to direct successive iterations. Some researchers propose tuning specific compiler functions or their order. Stephenson et al. [15] use genetic programming to optimize the specific heuristic associated with compiler optimization work. Cooper et al. [5] use a genetic algorithm to find the preferred order of optimization phases that generates smaller code in an embedded system environment. Kulkarni et al. [11] describe a solution for making an exhaustive search of the optimization phase order space. Bodin et al. propose a special iterative compilation system [4], which efficiently explores a large transformation space consisting of small number of optimization options (3), where each option is associated with a numerical parameter. Instead of exploring available optimization options, Frank et al., [6], suggests to have the iterative stochastic search explore the space of source-level transformation, thereby overcome the limitations imposed by fixed set of optimization options.

The large number of evaluations inherent in the iterative approaches is addressed by Agakov et al.[2]. Using trained predictive models their system is able to reduce number of iterations to as few as two. Fursin et al. [7] address this problem by exploiting stable program phases to test different versions of functions, instead of dedicating full runs for that.

For users of traditional compilation systems such as GCC, who have no control over phase order nor any knowledge of compiler internals, the above works do not provide a relevant solution. In this context, the only control the user has is over the set of optimization options. Not many works are available here, apart from the commercial applications [1], and PathOpt tuning tool of EKOPath [13]. Pinkers et al. [14] use orthogonal arrays (OA) to iteratively trim down the number of actual optimization options used. Though the number of iterations is small, the number of total compilations can be quite large. Nisbet [12] uses a genetic algorithm to select loop restructuring transformations in Fortran programs. This work come closest to our work with its approach to selection of optimization options. There are many differences, however, in the details of the algorithms, as discussed in Sect. 3.

3 Iterative Optimization and Tuning

3.1 The Problem

In modern compilers and optimizers, the set of possible optimizations is usually very large. Some of the options require an additional parameter, which makes the selection process even more complex. Selecting the optimal set for a given application is complex because optimization options do not necessarily add to performance when combined with other options, and/or when used for certain application and a certain workload. As a result of this complexity, users tend to use standard combination options, such as -O, -O2, and -O3. These combinations select a subset of the optimization options that are available to provide stable, high performance for a large number of applications. Thus, in general, these options are not the optimal set for a specific application executing its representative workload.

Table 1. Estimated Amounts of Optimization Options

Optimizer	Binary	Parameterized	Effective
FDPR-Pro	22	12	58
GCC	55	5	70
XLC	52	1 (see note)	55

Table 1 gives an estimate for the number of options in different optimizers. The exact assessment for GCC and especially for XLC is difficult, because some options have complex multilevel syntaxes and dependencies (Note: for that reason only one one parametrized option was configured for XLC). Generally, we can see that the problem is highly multidimensional, with many binary and multi-value/continuous parameter dimensions. To estimate the effective search space, we assume parameterized options have eight discrete possible values (which is conservative, see typical cases in [4]). Now the total number of option combinations becomes $2^B \cdot 8^P = 2^{B+3P} = 2^E$, where B and P are the number of binary and parameterized options respectively, and $E = B + 3P$ the number of effective binary options. Study of such problems [4] shows that the performance landscape in this highly multidimensional space is nonlinear with many local optima. The ability of genetic algorithms to search efficiently in such an irregular multi-dimensional space makes them a preferred choice.

3.2 Genetic Algorithm

Algorithm 1 ESTO specific GA implementation

```
configure optimizer options, GA parameters and *init* organism
create population of randomized organisms and one *init*
evaluate fitness of *init* organism
**for** $generation = 1$ to $generation.limit$ **do**
  evaluate fitness of all organisms
  sort all organisms by fitness
  print all organisms with gain relative to *init*
  **if** $generation.best.result$ has improved **then**
    $no.improvement.counter \Leftarrow 0$
  **else**
    $no.improvement.counter \Leftarrow no.improvement.counter + 1$
    **if** $no.improvement.counter = no.improvement.limit$ **then**
      terminate
    **end if**
  **end if**
  **if** $variable.population.size$ **then**
    reduce $population.size$
    **if** $population.size = 1$ **then**
      terminate
    **end if**
  **end if**
  replace lower population half by upper population half {natural selection}
  randomly blend 1st quarter organisms into 4th, and 2nd into 3rd {crossover}
  randomly mutate lower 3/4 of population with $option.mutation.rate$
  **if** $adaptive.policy$ **then**
    fully re-randomize all organisms worse than *init*
  **end if**
**end for**
```

Genetic Algorithm (GA) [17, 3] is a search technique inspired by evolutionary biology concepts such as inheritance, mutation, selection, and crossover (recombination). Optimization parameters are encoded as *genes* of separate *individuals*. Each individual's combination of parameters represents a possible candidate solution to the optimization problem. A *population* of individuals evolves during multiple *generations* toward better solutions. The evolution usually starts with a population of randomly generated individuals. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are selected based on their fitness, then mutated and recombined to form a new population. This process repeats until a termination condition is reached.

In our specific GA implementation, each gene represents an optimizer *option*, either *binary* (on/off), or with a parameter in a certain *range* of type *int, float, power2*, and *enum*. These options are assembled into *option sets* that correspond to the optimizer invocation arguments, typically specified in a single command line. Each option set is encapsulated in an *organism*, i.e., a GA individual. A number of organisms constitute a population, which evolves for a limited number of generations (see Algorithm 1). The evolution stops when there is no improvement of the best result for a few generations, or when the *generation limit* is exhausted, or when the population is reduced to a single organism (see Sect. 3.4 on population size alterations).

The crossover of genes and the natural selection are done by cross-breeding corresponding organisms from the upper two quarters of the population, and replacing the lower two quarters with the resulting "children". (*Cross-breeding* means that each option is separately inherited from either parent, with a given *crossover rate* probability for the lower parent. For an option with a parameter, its value is randomly dealt between the parents' values.) Then, the upper quarter is left untouched (*elitist selection*), and the lower three quarters are mutated with a given option-wide *mutation rate*, i.e., mutation probability for each option. (*Mutation* means dealing a new value to the option, irrespective of the old value.)

In our implementation, the initial random population is complemented by a pre-configured "initial" individual, which can reflect either the composition of a known-good combination like -O3, or a "zero" option set, or a previous search result, or some arbitrary user choice. This *init* organism serves as a kind of pivot for further comparisons and intermediate gain computations.

3.3 Efficient Use of Search Budget

As shown in Sect. 5, GA experimental results are quite good, but they are achieved by large number of optimization + measurement cycles, typically above one hundred per application. This could be a time-consuming investment of resources, especially with large real-life applications whose representative workloads might run for a long time. It is therefore important to maximize the efficiency of the GA search by using the given fixed search budget to reach the best possible performance gain (see [9]).

3.4 Search Beam Width

One way to use a given test budget more efficiently is by shaping GA population size as a function of the generation number. Basic GA keeps the number of individuals constant throughout the whole run. Our intuition is that extending the random coverage in the beginning, and focusing on convergence in the end, improves the cost/performance efficiency. To this extent, we can use the term *search beam width*, analogous to the typical searchlight operation, where in the beginning we use a wide searchlight beam to choose among many potential target areas, and then narrow it to focus on a specific promising area. The implemented searchlight width policies (i.e., population size alterations) appear in Fig. 1 and are described below. Note that the areas below each policy graph (i.e., total amounts of tests) are roughly same, corresponding to the *budget limit* mentioned in the title of this paper.

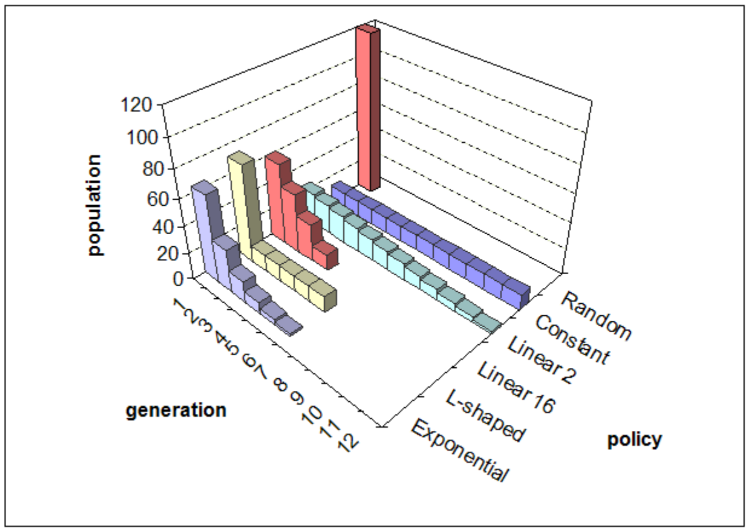


Fig. 1. Search beam width policies

Random – Pure random search, a single generation of 126 organisms + *init*. A trivial algorithm which serves as a sanity check for the proposed algorithms.

Constant – Like the above GA implementation with 12 generations of 12 organisms (total 144 tests), but without the "”reduce $population - size$" step.

Adaptive – Like *Constant*, but fully randomizes any organism with fitness below *init*. We regard this as search beam width policy, because "underperformers" from any generation are re-randomized, which likens them to individuals in the first generation and thus effectively reshapes the population size over generations. This technique was proved advantageous for the result gain and is used in the following methods.

Linear 2, 16 – Like *Adaptive*, but with linearly decreasing number of organisms in generations, either by 2: 24, 22, ... 2 (total 156), or by 16: 60, 44, 28, 12 (total 144). The dependence between the population size N and the generation number g is:

$$N(g+1) = N(g) - gradient \tag{1}$$

L-shaped – Like *Adaptive*, but with 72 organisms in the first generation and 12 organisms in the remaining 5 generations (total 132 tests).

Exponential – Like *Adaptive*, but with 6 generations containing exponentially decreasing numbers of organisms: 64, 32 ... 2 (total 126 tests). The dependence between the population size N and the generation number g is:

$$N(g+1) = N(g)/2 \tag{2}$$

Note that this can be expressed in absolute (rather than iterative) terms, using the generation limit L:

$$N(g) = 2^{L-g+1} \tag{3}$$

Our hypothesis is that the *Exponential* policy is the most efficient one. It should have the best cost/performance ratio and thus utilize the budget limit in the best possible way. The rational behind this hypothesis is the following: Each passing generation decreases the *uncertainty* that we indeed found the optimum. It seems plausible that this decrease is roughly the same as that of the previous generation, so that the uncertainty decreases exponentially as function of the generation number. Thus, we can reduce the population size exponentially as well, without harming the above uncertainty decrease rate, akin to the per-generation convergence rate.

4 Configurability

ESTO can be used for a wide variety of optimization tools. It has already been applied successfully to GCC and XLC, as well as to FDPR-Pro, IBM feedback-directed post-link optimization tool [10]. It was adapted to Linux, AIX, and various embedded board setups, as well as to a number of benchmark configurations: direct application invocation, SPEC2000, and UMT2K [16].

Much of ESTO flexibility is due to its configuration file. The file consists of two sections: algorithm control and option specification (see Sects. 4.1 and 4.2). The latter section enables ESTO to regard the optimizing compiler as a black box, controlled solely through its user-visible optimization options.

4.1 Search Algorithm

The concrete *search algorithm modification* to use is specified in a configuration file line, and can be any of those described in Sect. 3.4, plus a few special investigative modifications. The algorithm can be configured with its own *arguments*, like *L-shaped* sizes, *Linear* gradients, and *Exponential* bases, and also these common GA parameters (defaults in parentheses): *mutation rate* (0.01), *crossing rate* (0.5), *initial population size* (12), *generation limit* (12), and *no improvement limit* (4).

4.2 Application

By the term *application*, we mean the whole range from the *option set* to the *reported time*, typically including the invocation of an *optimizer* on some *target program*, and measuring the resulting performance in an environment-specific fashion.

Optimizer Options – This configuration section specifies the set of options from which the optimal set should be selected. Each option line specifies its *name*, whether switched on in the *init* organism, and the *probability* of switching on upon mutation (e.g., can be set to the option occurrence rate in the historic ESTO results, to speed up convergence).

Two kinds of options can be specified: *binary* and *range* (parameterized). The latter, in turn, has a few types: *int, float, power2*, and *enum.* Each type should specify *low* and *high* boundaries of parameter value variations, an *initial* parameter value, and a parameter *step*, i.e., the natural or artificial parameter value alignment.

Application and Workload – The behavior of the default application and its workload is controlled by a few default scripts that can be provided by the user according to defined command interfaces. Alternatively, the user can extend and re-implement the application class interface.

Most importantly, the *application script* runs the user application with its required parameters, including an *option set* whose fitness has to be measured, and *transit arguments*, passed directly and transparently from ESTO command line invocation. There are a couple of other default command interfaces for error handling and synchronization.

As it runs, ESTO prints to standard output the performance results of each organism, and the sorted summary of each generation. Finally, it outputs the result line, which lists the chosen option set, its reported time, and the gain percentage.

4.3 Multiplexor

The *multiplexor* interface implements scattering of the *application* trials, gathering of their results, and synchronization as requested by the *algorithm*. ESTO supports single-threaded mode and a parent/workers style multiprocessing (default). Other multitasking implementations are possible: multithreading, grid or blade architecture, clusters, etc. The interface is also responsible for fault tolerance: identifying, getting rid of, and recovering from failing runs. The task is particularly important here because arbitrary selecting of options may bring compilers to untested regions and possible failures.

5 Experimental Results

5.1 Metric

Typically, maximizing an application's performance means either reducing its running time on a predefined workload, or performing more operations (processing larger workload) during a given time. We use running times, so our fitness metric is descending – the shorter is the running time, the better is the performance.

All of our experiments were done with SPEC2000 benchmark suite (*train* workloads) on Linux machines with Power architecture. ESTO invokes the user application script as mentioned in Sect. 4.2. That script typically invokes `runspec` as part of its operation, and then extracts results from the `reported_time` fields of the raw result files. These results are then piped back to ESTO for comparison inside GA. Detected failures are considered as `FLT_MAX`.

5.2 Comparison of Results for Different Optimizers

Although relatively inefficient, the *Constant* policy allows us to compare ESTO achievements for different optimizers relative to various starting points. This is due to the abundance of available historic results on machines of same type: 4-processor 1.5GHz Power5 running SUSE Linux Enterprise Server 9 operating system. The GA parameters were also identical: mutation rate 0.02, crossover rate 0.45, population size 12, generation limit 12, no-improvement limit 4.

GCC – ESTO-on-GCC setup includes GCC-specific SPEC2000 configuration and GCC optimization options configuration for ESTO (see Sect. 4.2). The *init* seed organism corresponds to the composition of -O1 combination. On most benchmarks ESTO obtains high gains over -O1 starting point. Even relative to -O3 these are reasonably good gains in half of the cases. If we would start from -O3 seed, ESTO results would not go below the -O3 result. See Fig. 2.

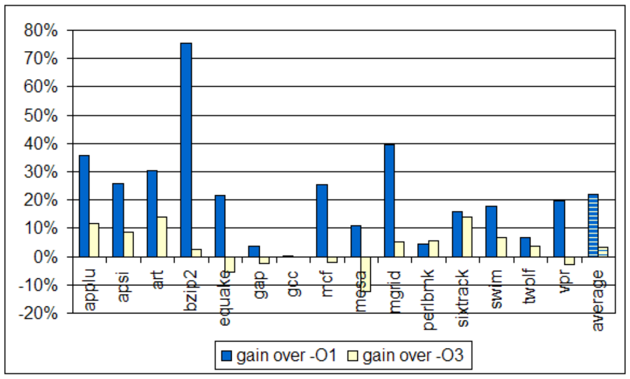


Fig. 2. ESTO gain over GCC -O1 and -O3

XLC – ESTO-on-XLC setup includes XLC-specific SPEC2000 configuration and XLC optimization options configuration for ESTO. The *init* seed organism has all options switched off (*zero* organism). On most benchmarks ESTO obtains high gains over -O3, with average of 13%. We compared also to the *peak* option set results reported[1] to SPEC for XLC on Power5 with SLES9. Most of these reports include XLC profile-driven feedback facility, so we extended ESTO application setup to run profiling phases, which prolongs optimization times significantly. As can be seen in Fig. 3, in some cases ESTO gains even over these, certifiably the best reported, results.

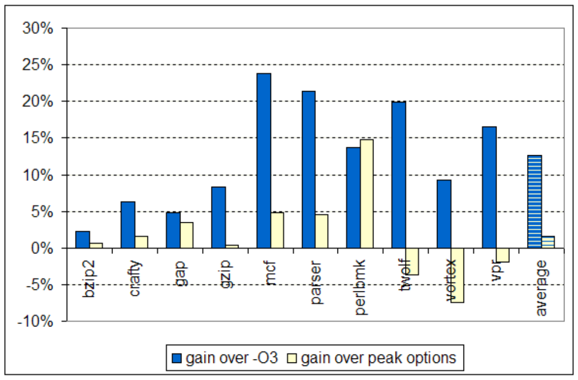


Fig. 3. ESTO gain over XLC -O3 and over peak option sets per benchmark

[1] www.spec.org/osg/cpu2000/results/res2004q4/cpu2000-20041018-03448.html

FDPR-Pro – ESTO-on-FDPR-Pro setup includes FDPR-Pro-specific SPEC2000 configuration with feedback-directed optimization stages and FDPR-Pro optimization options configuration for ESTO, which includes relatively large number of parameterized options. The *init* seed organism corresponds to the composition of -O3 combination. On most benchmarks ESTO obtains high gains over -O3 starting point, see Fig. 4. One interesting question in SPEC context is whether the option sets found by ESTO on the *train* workload can benefit the *ref* workload (the formal reference workload of SPEC) . In Fig. 4 the left set of bars contains ESTO gains over FDPR-Pro -O3 on *train* workload, and the right set depicts *ref* gains over -O3 with the option set found on *train.* In most cases there seems to be a significant correlation between the two, so *train* could be used as a predictor for *ref.* The practical significance of this finding stems from the fact that *train* runs some order of magnitude faster than *ref.* For the *ref* workload the above hundred tests, required by ESTO's, might be prohibitively slow, but can complete in reasonable time on *train.*

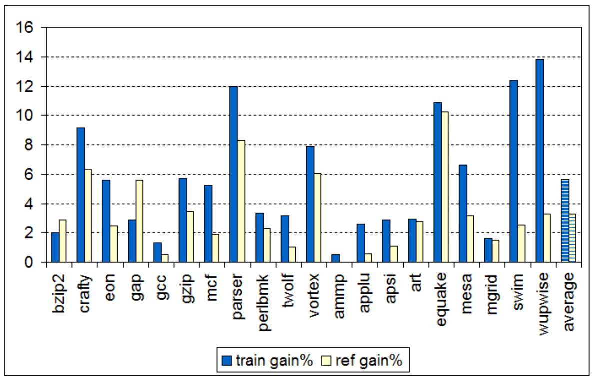


Fig. 4. ESTO gain over FDPR-Pro -O3 found on *train* workload, and *ref* workload gain with same option set

Figure 5 depicts the typical course of ESTO run on an example: *mesa train.* The upper line shows how the best gain in generation approximates the final result. The vertical range bars denote the distribution range of the upper half of that generation's individuals, in line with our GA natural selection implementation. We clearly see the reverse-exponential decrease of both the result approximation and the distribution range. This provides visual experimental support to our hypothesis that the uncertainty of the result decreases exponentially, with a constant ratio (convergence rate) between two successive generations.

Convergence Comparison – To assess the rate of convergence, we use the following convergence criterion: Two thirds of the population reached gain within 5% of their mean. Using this criterion on the accumulated ESTO logs of the

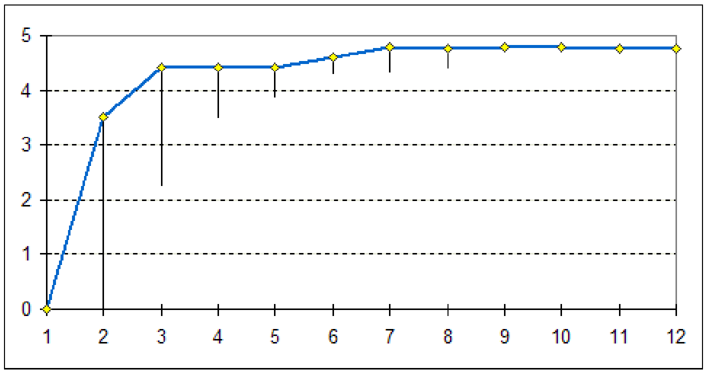


Fig. 5. ESTO best gain % and distribution range in each generation

above experiments, we measured the following average number of test iterations required for optimizers to converge:

– GCC – 119
– XLC – 59
– FDPR-Pro – 94

GCC's comparatively slow rate can be attributed to its large amount of options (see Table 1) with performance benefits more evenly spread between option combinations (wide distribution). XLC's fast convergence seems to be caused by a few very dominant option combinations which greatly affect performance, while others barely matter (narrow distribution). Thus the *effective* amount of options is quite smaller than Table 1 suggests. FDPR-Pro is somewhere in the middle in both senses – convergence rate and options benefits distribution.

5.3 Comparison of Population Size Policies

ESTO GA policy comparative study was conducted on a single-core Power970 2.2GHz blade machine with SUSE Linux operating system. The study compares the policies discussed in Sec. 3.4: *Random*, *Constant*, *Adaptive*, *Linear 2/16*, *L-shaped*, and *Exponential*. Fig. 6 shows the results of this study: ESTO *total* and *per-iteration* gains over FDPR-Pro -O3 when using different policies.

Total Gain – Represented by wide bars in Fig. 6.

There is a big jump from *Constant* to *Adaptive*, probably due to a better search space coverage created by re-randomizing of many "underperformers". Both *Linear* policies seem to interfere with this consideration by artificially reducing that coverage. *L-shaped* and *Exponential* are obviously less affected, perhaps because both rapidly get rid of those "underperformers" anyway. We see that *L-shaped* and *Exponential* policies reach maximum total performance gain.

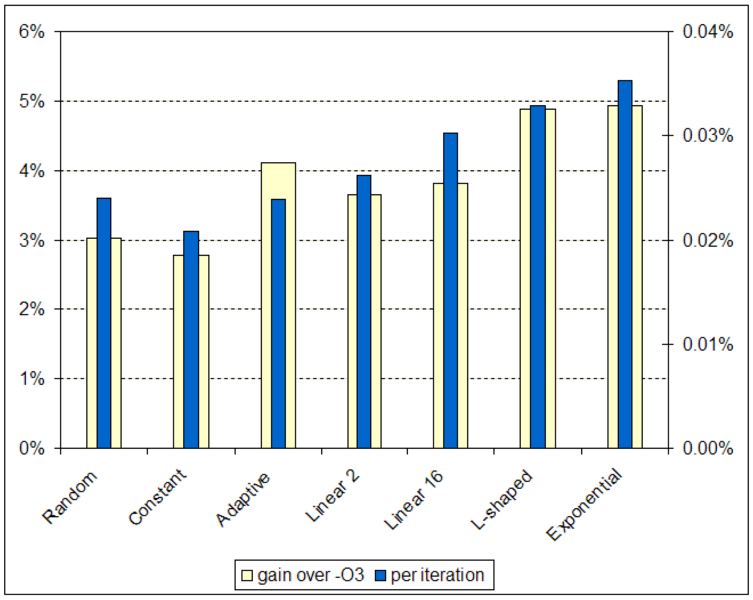


Fig. 6. ESTO total and per-iteration gain over FDPR-Pro -O3 with different policies

The superiority of *Random* over *Constant* can be explained by *Constant*'s apparent "wasting" of the budget. In each generation, *Constant*'s iterations above *Exponential* (with same size of generation 1) – are redundant according to our hypothesis. *Random*, on the other hand, does not "waste" budget, it just does not "complete" the exponential tail befittingly with its only generation 1. See also [8].

Gain per Iteration – Represented by narrow bars in Fig. 6.

Notably, we see the gain rate steadily rising as the policy shape approaches the decreasing exponential curve. This algorithm efficiency indicator culminates when using the *Exponential* policy, as suggested by our hypothesis in Sect. 3.4.

5.4 Parallelization Study

Reduction of ESTO running time is an important technical challenge, because it improves response time and makes more efficient use of resources. ESTO total processing task for each SPEC2000 benchmark consists of multiple iterations of FDPR-Pro optimization and *train* workload measurement. The total duration of sequential ESTO runs on all benchmarks in the suite reached some 5-6 days on a 1.5GHz Power5 machine. Durations of the iterations' stages vary widely between benchmarks, from 16 seconds optimization for *parser* and 3 seconds measurement for *gcc*, to as much as 12 minutes optimization for *perlbmk* and 1.5 minute measurement for *ammp*. Overall, when *train* workload is used for measurements (25 seconds on average), the optimization stages consume 3/4 of the iteration time (1.5 minutes on average). Of course, *ref* workload would have reversed this situation.

Superficially, we may want to evaluate the organisms of a generation in parallel. An important constraint here is that the measurement stages generally should not run in parallel on the same machine, because of the shared L2/L3 cache. The approach taken is then to perform the optimization stages of a generation in parallel, while the measurement stages are done serially. This allowed to reduce the above running time by half, using the machine's 4 cores (2 HW threads/core, i.e. 8 "logical" CPU).

We then studied the effect on measurement accuracy when this phase is done in parallel. The machine used was 1.5GHz Power5 system with 8 cores (2 HW threads/core, i.e. 16 "logical" CPU). The assumption was that for single-threaded medium-size applications, like the SPEC2000 suite, with one process per core (to avoid interference between HW threads of the same core), measurements will not interfere with each other. This was verified as shown in Fig. 7. Up to N parallel measurements on an N-core machine ($N = 8$) practically do not interfere, and above N there is a linear degradation with gradient about $1/2N$ per process addition.

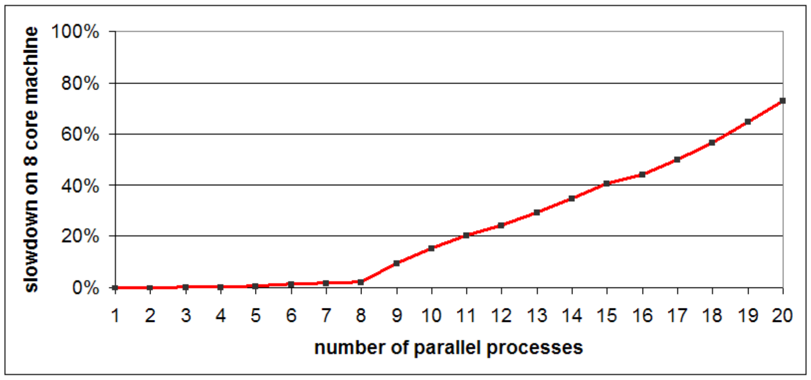


Fig. 7. Slowdown of SPEC2000 measurements (median) when parallel processes run the same benchmark

In our case, running ESTO on all benchmarks using 6 cores (12 logical CPU) out of the available 8 (16), while parallelizing *both* optimizations and measurements – completed in 16 hours only. In practical terms, this means that instead of waiting for the ESTO result for a working week, one can get the result overnight, thus greatly increasing the tool's usability.

Such technique can be used for single-threaded applications only. Parallelizing the tuning of multitasking applications calls for distribution of GA organisms between different machines.

6 Conclusions and Future Work

We show the feasibility of using the genetic algorithm for selecting preferred optimization options for a variety of compilers and workload, dealing with more then more then 60 effective options, and converging to the preferred result in around 100 iterations, depending on the option space. While we do not have formal proof, the stability of the results, when starting from an arbitrary organism, show that the preferred results are close to the optimum. This is comparable or better to the other alternatives discussed in this paper. The convergence speed, in gain/iteration, is a critical factor in iterative compilation. We studied a number of search beam policies and showed that an exponentially decreasing beam gives the best result.

The database of past results associates a specific workload to recommended option sets. We plan to exploit this database by using automatic feature selection and clustering to acquire quick prediction of the recommended option set for a given workload, or a recommended starting point for the GA search.

References

1. ACOVEA. Available at http://www.coyotegulch.com/products/acovea/index.html, 2006.
2. F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the 4thAnnual International Symposium on Code Generation and Optimization (CGO)*, March 2006.
3. E.B. Baum, D. Boneh, and C. Garrett. Where genetic algorithms excel. *Evolutionary Computation*, 9(1):93–124, 2001.
4. F. Bodin, T. Kisuk, P. M. W. Knijnenburg, M. F. P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback-Directed Compilation, in Conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, October 1998.
5. Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '99)*, pages 1–9, New York, NY, USA, 1999. ACM Press.
6. B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded systems software. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, pages 78–86, June 2005.
7. Grigori Fursin, Albert Cohen, Michael O'Boyle, and Oliver Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the 1st International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2005)*, pages 29–46, November 2005.
8. G.G.Fursin, M.F.P.O'Boyle, and P.M.W. Knijnenburg. Evaluating iterative compilation. In *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC'02)*, 2002.

9. A. Grajdeanu and K. A. De Jong. Fixed budget allocation strategy for noisy fitness landscapes. In *Late Breaking Papers of the Genetic and Evolutionary Computation Conference (GECCO-2003)*, 2003.
10. G. Haber, E. A. Henis, and V. Eisenberg. Reliable post-link optimizations based on partial information. In *Proceedings of Feedback Directed and Dynamic Optimizations 3 Workshop*, pages 91 – 100, December 2000.
11. A. P. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson. Exhaustive optimization phase order space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.
12. A. Nisbet. Gaps: Genetic algorithm optimised parallelisation. In *Proceedings of the 7th Workshop on Compilers for Parallel Computing*, 1998.
13. PathOpt. http://www.pathscale.com/pdf/PathOpt2-paper.pdf.
14. R. P. J. Pinkers, P. M. W. Knijnenburg, M. Haneda, and H. A. G. Wijshoff. Statistical selection of compiler options. In *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS '04)*, pages 494–501, Washington, DC, USA, 2004. IEEE Computer Society.
15. M. Stephenson, U. O'Reilly, M. Martin, and S. Amarasinghe. Genetic programming applied to compiler heuristic optimisation. In *Proceedings of The 6th European Conference on Genetic Programming*, 2003.
16. The UMT benchmark code. Available at http://www.llnl.gov/asci/platforms/purple/rfp/benchmarks/limited/umt/, 2002.
17. D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, 6 1994.

Pruning the Optimization Search Space Using Architecture-aware Cost Models*

Apan Qasem   Ken Kennedy

Department of Computer Science
Rice University
Houston, TX
{qasem,ken}@cs.rice.edu

Abstract. In recent years, a number of strategies have emerged for empirically tuning applications to different architectures. Although quite successful for certain domains, empirical tuning is yet to gain wide acceptance as a viable strategy in high-performance computing. The principal bottleneck in this regard is the prohibitively large search space that needs to be explored in order to discover the best program variants for different architectures. Although there have been some efforts at using cache models in pruning the search space for kernels, the optimization search space of whole applications still remains mostly intractable. In this paper, we propose a novel search space pruning strategy. Our approach is to identify architecture-dependent parameters within compiler cost models and search for the best values of those parameters. We have implemented this strategy for exploring the search space of loop fusion and tiling for whole applications. Preliminary experiments suggest that this approach of tuning for architecture-dependent model parameters is highly effective in reducing the size of the optimization search space while incurring only a small performance penalty.

1 Introduction

Over the last several decades we have witnessed tremendous change in the landscape of computer architecture. New architectures have emerged at a rapid pace and at the same time, the complexity of microprocessor architecture has grown consistently. The changing nature of the processor architecture and its ever increasing complexity, has made retargeting of applications a major concern for high-performance computing. The advent of each new architecture and even a new model of a given architecture has required retargeting and retuning of applications at considerable cost. To address this issue, many strategies for automatic tuning have been proposed [9, 4, 5, 8]. Although the empirical approach has been quite successful in generating highly-tuned domain specific libraries, its application in tuning general scientific programs has been limited. The chief obstacle

* This material is based on work supported by the Department of Energy under Contract Nos. 03891-001-99-4G, 74837-001-03 49, 86192-001-04 49, and/or 12783-001-05 49 from the Los Alamos National Laboratory.

in this regard is the prohibitively large search space that needs to be explored to find optimal transformation parameters. For example, Zhao et al. [13] show that the search space for fusing n statements into m loops without any reordering can be as large as $\binom{n-1}{m-1}$. Clearly, exploring such a large space is infeasible for a general-purpose compiler. Recent papers advocate model-guided tuning as a means of pruning this enormous search space [11, 6, 3]. In the model-guided approach, analytical models are used to restrict the search space to regions that are likely to contain mostly good values.

In this paper, we propose a new approach to pruning the optimization search space. In our strategy, we move away from the search space of parameterized transformations and instead focus on the search space of architecture-dependent parameters embedded within the cost models. As we know, the profitability of many program transformations are sensitive to certain machine parameters. For example, tile sizes are constrained by the capacity of the target cache. Compiler cost models use these architectural parameters as a means for picking the best transformation parameters. However, in most cases these parameters are difficult to determine accurately. For example, the fraction of cache we can exploit depends on the size and associativity of the cache, the number of different arrays we access in the program and also the size of each of those arrays. A static model that attempts to capture all these parameters is unlikely to be totally accurate for all architectures. The goal of our tuning strategy is to correct for these inaccuracies in the cost model. We use empirical search to find the best estimates of the machine parameters which in turn deliver the best set of transformation parameters.

Our pruning strategy reduces the size of the search space in two ways. Firstly, we can use a single parameter to capture the effects of multiple transformations which reduces search space dimensionality. For example, we can use the estimate of the cache size parameter to tune both loop fusion and tiling parameters. Secondly, for transformations that can have different parameters for different loops (i.e. tiling), we can again use just a single parameter to tune each of the loops in the program. Thus, the search space we explore does not grow with program size. For large applications with many loop nests, this property can be very effective in limiting the size of the search space.

2 Related Work

In recent years, there has been a flurry of work in empirical tuning that aim to improve the tuning process using machine learning, statistical methods and heuristic search strategies. However, relatively few of these have addressed the issue of using compiler models in pruning the optimization search space.

Knijnenburg et al. [6] introduced the notion of search space pruning using compiler models in the context of iterative compilation. In their work, they examine the effects of cache models on empirically tuning tiling and unroll factors. They use static models in combination with a cache simulator to filter out bad candidates with high cache miss rates from the parameter search space. Their results show that the use of cache models can indeed speedup the tuning process significantly without a high sacrifice in performance. An interesting and important aspect of this work is the use of *slack factors* to estimate the capacity of set-associative caches. These *slack factors* are determined experimentally and then used as a fixed value during the tuning process. In our work, it is these *slack factors* that form the basis of our search space. As we show in Section 3, having the *slack factors* integrated into the tuning process can significantly reduce the optimization search space.

Yotov et al. [11] show that analytic modeling alone can deliver performance that is comparable to that of ATLAS. In subsequent work, they have shown that modeling, combined with local search and model refinement is highly effective in generating optimized code for BLAS on different architectures [12]. Chen et al. [3] combines analytical models with empirical search to automatically tune dense matrix computations to two different architectures. They use static models to generate a parameter search space that is likely to contain the optimal parameter value. By combining their cache-conscious models with empirical search, they are able to achieve performance comparable to that of ATLAS on the matrix multiply kernel. The search process is about 2-4 times faster than that of ATLAS. Most recently, Agakov et al. [1] have used predictive modeling techniques to focus search strategies to more profitable regions within the search space. Their approach was highly effective in reducing the tuning time for a large search space of 82^{20} points.

Our approach to search space pruning is distinct from previous work mentioned in this section, in that we aim to tune architectural parameters integrated in our cost model rather than the space of transformation parameters.

3 Approach

Our approach to defining and pruning the optimization search space is best described as a three-step process. In this section, we describe each step in some detail.

Step 1: Identify architectural resources that affect profitability.

Most transformations - particularly those worth tuning for - are considered to be architecture sensitive; that is, their profitability depends on certain parameters of the target architecture. In most cases, the architectural parameters will impose constraints on the transformation parameters. For example, the profitability of unroll-and-jam is constrained by the register pressure of the unrolled loop [2]. The literature on code-improvement transformations is replete with many such examples. The first step in our tuning process is to identify key architectural resources that affect the profitability of the transformations in question and then determine the relationship between the architectural parameters and the transformation parameters.

Step 2: Construct parameterized models to estimate available resources.

Once the architectural resources have been identified, we need a mechanism to estimate the amount of resource that is available to the program. The amount of resource that can be exploited by a program is determined by a host of factors. For example, the fraction of cache we can exploit depends on the size and associativity of the cache, the number of different arrays we access in the program and also the size of each of those arrays. Hence, the second step in our tuning process is to construct models that estimate the amount of resource that is available. For each resource R, we construct a function that computes the *effective size* of R.

$$R' = \mathit{EffectiveSize}(r_1, r_2, ..., r_n) \ s.t. \ R' \leq R$$

where $r_1, r_2 ... r_n$ are parameters that determine the effective size of R.

Again, there is published work describing many such models. However, the key issue in using these models is finding a suitable parameterization, so that we can expose the relevant parameters for tuning through empirical search. We accomplish this by introducing the notion of a *tolerance term*. We derive tolerance terms for each of the machine parameter estimates such that there is a linear relationship between the tolerance term and the architectural resource. For example, we use the cache miss rate of the program as tolerance for estimating the effective cache capacity.

$$L'_k = \mathit{EffectiveCacheSize}(s_k, a_k, T)$$

where L'_k is the effective size of the cache at level k, s_k and a_k refer to the size and associativity of the cache and T is a tolerance term that corresponds to the miss rate at L_k.

Step 3: Search for best estimates using tolerance values.

Our search space is the Cartesian product of the sets of tolerance values used in estimating each architectural parameter. As such, any multidimensional search strategy used to explore the search space of transformation parameters can be effective in exploring the search space. However, a natural choice for exploring this search space turns out to be a sequential search. For each tuning parameter in the search space, we start off conservatively with a low tolerance value and increase the tolerance at each subsequent iteration. We stop the iterative process either when performance degrades or when we have reached the availability threshold of a particular resource. The rationale behind choosing a sequential search is the following: since at each step we allow the program to consume more of a particular resource, at some iteration we will reach a threshold value where the program will have consumed too much of that resource. From that point on consuming more of that particular resource will only further degrade performance. As we shall see in the experimental results in Section 5, this simple search strategy works fairly well for tuning loop fusion and tiling parameters.

4 An Example

In this section, we demonstrate the effectiveness of our strategy using a simple example. For this example, we only consider fusion of innermost loops and tuning of the effective register set parameter. We first explain the fusion parameter search space, then the search space for the effective register set and then present results from an experiment comparing the two search spaces.

If reordering of loops is not allowed, the number of different ways to fuse k loops is 2^{k-1}. Thus, the number of points in the fusion search space of k loops is 2^{k-1}. We can represent the search space of different fusion configurations using Gray Code ordering. In a Gray Code ordering a fusion configuration is represented using a bit pattern where each bit corresponds to an edge between two fusible loops. A bit is set if the corresponding adjacent loops are fused. For example, if we have eight fusible loops then we need bit strings of length seven where bit string `0000000` corresponds to no loops being fused and `1111111` corresponds to all loops being fused.

A key consideration for fusing loops at the innermost level is the register pressure of the fused loop. If the number of registers required to execute the fused loop is more than the number of available registers then fusion is unlikely to be profitable because of register spills. For this reason, a compiler cost model for fusion will usually impose the following constraint for fusing loops:

$$\textit{Register Pressure}(\textit{Loop}_{fused}) < \textit{Effective Register Set}$$

where *Register Pressure* is the number of registers required to execute the fused loop and *Effective Register Set* is the number of physical registers available to the loop at runtime.

Although there are several algorithms that can estimate the register pressure of a loop nest, the actual number of physical registers available at runtime is usually much more difficult to determine accurately. Clearly, the above constraint in the cost model will be ineffective if we do not have an accurate estimate of the effective register set. Conversely, the constraint will produce the best results when we have the best estimate for the effective register set. Hence, in our empirical tuning framework, we search for the best estimate of the effective register set with the assumption that the best estimate will generate the best fusion configuration.

The number of available registers at runtime is a subset of the actual number of physical registers in the program. Hence, we estimate the effective register set using the following formula:

$$\textit{Effective Register Set} = \lceil T \times \textit{Register Set Size} \rceil, \quad 0 < T \leq 1$$

Here, T is the *tolerance term* that determines the fraction of the physical register set that is available to us at runtime. Thus, at each step in our search process we generate and evaluate a fusion configuration where the register pressure of each fused loop is less than the effective register set for some value of T. For example, if $T = 0.5$ and the number of physical registers is 64 then we will generate a fusion configuration where the register pressure of each fused loop is less than 32.[1] Thus, the search space for tuning the register set parameter is a set of tolerance values. The number of points in the search space is determined by how finely we wish to tune the parameter. For example, if we increase our tolerance by 0.05 at each step then we will have just 20 points in the search space. Note, that if increasing our tolerance does not result in a larger effective register set or a different fusion configuration then that point in the search space does not need to be evaluated. Thus, the number of points in the search space is bounded above by the size of the register set of the target platform and in practice, the number of points that need to be evaluated is likely to be much smaller than this upper bound.

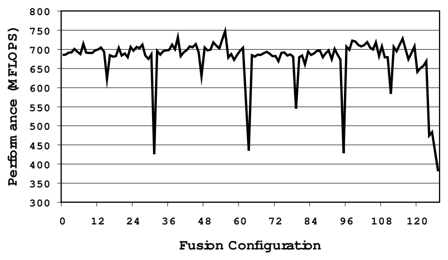


Fig. 1. Performance curve for fusion configuration search space on Opteron

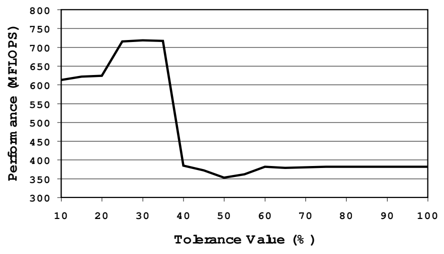


Fig. 2. Performance curve for effective register set search space on Opteron

To compare the two different search spaces we perform a simple experiment with the `advect3d` kernel from the NCOMMAS [10] weather-modeling application. The `advect3d` kernel has a total of 24 loops divided into eight loop

[1] Note, for any given tolerance term there can be multiple fusion configurations. Our static cost model determines which of those configurations will be picked for evaluation.

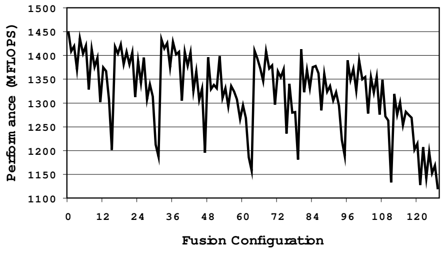


Fig. 3. Performance curve for fusion configuration search space on Pentium 4

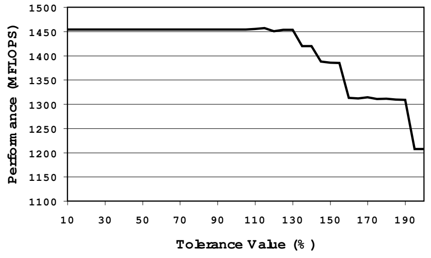


Fig. 4. Performance curve for effective register set search space on Pentium 4

nests which are perfectly nested. All loop nests are fully fusible. For this experiment, we consider fusing only the innermost loops without any reordering. As explained previously, the fusion search space for `advect3d` contains $2^{8-1} = 128$ points. The size of the search space of the register set parameter is dependent on the tolerance value increments and the number of physical registers in the target platform. We present performance results for these two search spaces on two platforms: a 2 GHz Opteron with 32 floating-point registers and a 2.4 GHz Pentium 4 with 8 floating-point registers. For both platforms, we increase tolerance by 5% at each step. Hence, for both platforms, the register set search space contains 20 points. However, since the number of registers on Pentium 4 is less than 20, the number of points that result in different fusion configurations is bounded above by the number of physical registers.

The performance of all possible fusion configurations on the Opteron is shown in Fig. 1. As expected, the performance line is very jagged with many peaks and valleys. The performance curve for the effective register set search space on the same platform is shown in Fig. 2. As explained earlier, this search space is much smaller than the search space of fusion configurations. However, the important thing to note here is that the performance line for this search space is relatively *smooth*. Not only that, the performance line follows a specific pattern. Initially, when we increase the tolerance from very low values (i.e. 10%) performance keeps increasing. Then, when $T = 35\%$, there is a big drop in performance. According to our search heuristic, $T = 35\%$ represents the threshold point and no further exploration of the search space is necessary. Indeed, we observe that none of the points beyond this threshold produce better performance. Hence, we could stop our search after evaluating just seven points in this search space. Another issue to note, is the leveling-off of the tail-end of the performance curve. This happens because all eight loops in `advect3d` are fused at the 55% tolerance level and the fusion configuration does not change for any value of T beyond that point. Hence, even if we were doing an exhaustive search we would not need to evaluate this portion of the search space.

The performance curves for `advect3d` on Pentium 4 are presented in Figs. 3 and 4. We notice very similar results on this platform as well. A jagged performance line for the fusion configuration search space and a smooth line for the search space of the effective register set parameter. Since Pentium 4 has so few floating-point registers, only a single pair of loops is fused when we increase our tolerance to a 100%. This explains the long flat segment at the beginning of the performance line in Fig. 4. Our search heuristic does not evaluate points beyond the 100% threshold. Hence, the search on this platform stops at $T = 100\%$ after fusing just one pair of loops. To verify that this conservative approach is indeed the right one, on this platform, we forced the search strategy to evaluate points beyond $T = 100\%$. As the results in Fig. 4 show, going beyond the maximum threshold and trying to fuse more loops makes the performance worse. Thus, for this platform it is best to stop at $T = 100\%$.

5 Experimental Results

We have implemented our search space pruning strategy for two transformations: loop fusion and tiling [7]. Our search space consists of tolerance values of two architecture-sensitive model parameters: *Effective Register Set* and *Effective Cache Capacity*. Our testing platform is a MIPS Origin r12000. We select four programs that exhibit opportunities for loop fusion and tiling: `advect3d`, an advection kernel for weather modeling, `erle`, a differential equation solver, `liv18`, a hydrodynamics kernel from Livermore loops, and `mgrid`, a multi-grid solver from SPEC 2000. For each program we run two sets of experiments: one using a sequential search on the pruned search space (`model-based`) and another using a multi-dimensional direct search strategy on the *un-pruned* search space (`direct`).

Performance results from four applications on the MIPS are presented in Fig. 5. The results show that `model-based` is able to find values that are very close to the values found by `direct`. The performance gap is never more than 5%. On the other hand, in terms of tuning time we pay a high premium when we apply direct search. Fig. 6 shows that on average `direct` requires about four times as many program evaluations as `model-based`. In the context of empirical tuning, where number of program evaluations is the principal bottleneck, this savings in tuning time will be significant for any decent-sized application. In such cases, the savings in tuning cost will make the small sacrifice in performance worthwhile.

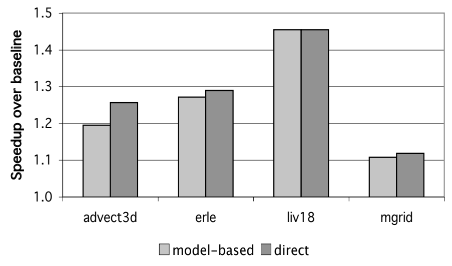


Fig. 5. Performance comparison: `model-based` vs `direct`

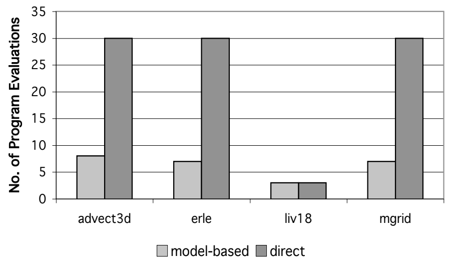


Fig. 6. Tuning time comparison: `model-based` vs `direct`

6 Conclusions

In this paper, we have presented a method of pruning the optimization search space by searching for architecture-dependent model parameters. Preliminary experimental results suggest that this approach can be highly effective in reducing the size of the search space while incurring only a small performance penalty.

Acknowledgement. We would like to thank the anonymous reviewers for their constructive comments and helpful suggestions in improving the quality of this paper.

References

1. F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization, 2006. (CGO 2006).*, New York, NY, 2006.
2. S. Carr. *Memory-Hierarchy Management.* PhD thesis, Dept. of Computer Science, Rice University, Sept. 1992.
3. C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization*, San Jose, CA, 2005.
4. K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21^{st} century. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, Oct. 2001.
5. G.G.Fursin, M.F.P.O'Boyle, and P.M.W.Knijnenburg. Evaluating iterative compilation. In *Proceedings of the Fifteenth International Workshop on Languages and Compilers for Parallel Computing*, College Park, Maryland, July 2002.
6. P. Knijnenburg, T. Kisuki, K. Gallivan, and M.F.P.O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. *Concurrency and Computation: Practice and Experience*, 16:247–270, 2004.
7. A. Qasem and K. Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th ACM International Conference on Supercomputing*, June 2006.
8. S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, San Fransisco, CA, 2003.
9. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98: High Performance Networking and Computing*, Orlando, FL, Nov. 1998.
10. L. J. Wicker. NSSL collaborative model for atmospheric simulation (NCOMMAS). `http://www.nssl.noaa.gov/~wicker/commas.html`.
11. K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.
12. K. Yotov, K. Pingali, and P. Stodghill. Think globally, search locally. In *Proceedings of the 19th annual international conference on Supercomputing (ICS06)*, 2005.
13. Y. Zhao, Q. Yi, K. Kennedy, D. Quinlan, and R. Vuduc. Parameterizing loop fusion for automated empirical tuning. Technical report, Lawrence Livermore National Laboratory, Dec. 2005.

Building a Practical Iterative Interactive Compiler

Grigori Fursin and Albert Cohen

ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud University, France
{grigori.fursin,albert.cohen}@inria.fr

Abstract. Current compilers fail to deliver satisfactory levels of performance on modern processors, due to rapidly evolving hardware, fixed and black-box optimization heuristics, simplistic hardware models, inability to fine-tune the application of transformations, and highly dynamic behavior of the system. This analysis suggests to revisit the structure and interactions of optimizing compilers. Building on the empirical knowledge accumulated from previous iterative optimization prototypes, we propose to open the compiler, exposing its control and decision mechanisms to external optimization heuristics. We suggest a simple, practical, and non-intrusive way to modify current compilers, allowing an external tool to access and modify all compiler optimization decisions.

To avoid the pitfall of revealing all the compiler intermediate representation and libraries to a point where it would rigidify the whole internals and stiffen further evolution, we choose to control the *decision* process itself, granting access to the only *high-level features* needed to effectively take a decision. This restriction is compatible with our fine-tuning and fine-grained interaction, and allows to tune programs for best performance, code size, power consumption; we also believe it allows for joint architecture-compiler design-space exploration.By exposing only the decisions that arise from the opportunities suggested by the program syntax and semantics and only when the associated legality checks are satisfied, we dramatically reduce the transformation search space.

We developed an Interactive Compilation Interface (ICI) with different external optimization drivers for the commercial open-source PathScale EKOPath Compiler (derived from Open64); this interface is being ported to the GCC. This toolset led to strong performance improvements on large applications (rather than just kernels) through the iterative, fine-grain customization of compilation strategies at the loop or instruction-level; it also enabled continuous (dynamic) optimization research. We expect that iterative interactive compilers will replace the current multiplicity of non-portable, rigid transformation frameworks with unnecessary duplications of compiler internals. Furthermore, unifying the interface with compiler passes simplifies future compiler developments, where the best optimization strategy is learned automatically and continuously for a given platform, objective function, program or application domain, using statistical or machine learning techniques. It enables life-long, whole-program compilation research, without the overhead of breaking-up the compiler into a set of well-defined compilation components (communicating through persistent intermediate languages), even if such an evolution could be desirable at some point (but much more intrusive). It also opens optimization heuristics to a wide area of iterative search, decision and adaptation schemes and allows optimization knowledge reuse among different programs and architectures for collective optimizations.

1 Introduction

Iterative compilation is a popular approach for optimizing programs for different objective functions on architectures with ever growing complexity, when traditional compilers fails to deliver the best possible performance. Bodin et al. [7] and Kisuki et al. [24] have initially demonstrated that exhaustively searching an optimization parameter space for small kernels can deliver considerable performance improvements in comparison with state-of-the-art, single-run compilers. Cooper et al. [11, 12, 10] and later Kulkarni et al. [25] demonstrated that finding optimal optimization order can also considerably improve code quality and performance. We demonstrated hill-climbing and random iterative search techniques to optimize large applications on a loop-level in [18]. Later Triantafyllis et al. [37, 36] suggested a pruning technique to considerably speed-up optimization heuristic on a fine-grain level inside a compiler. Heydemann et al. [22] used iterative compilation to find trade-off between code size and performance improvement when using loop unrolling and code compression.

Iterative optimization has also been employed in well-known library generators in such systems as ATLAS [38], FFTW [26] and SPIRAL [32] which tune parameters of various transformations to get best performance on a targeted platform. Yotov et al. [39] and Epshteyn [13] use analytical model-based approaches to optimize BLAS libraries.

Many recent results address the iterative tuning of compiler flags, targeting performance or code size for a variety of applications [31, 20, 28–30, 14, 21]. Some of these techniques are already used by companies internally to tune the final settings of their compilers or even available to end-users such as PathOpt tool from the PathScale EKOPath compiler suite [2] that is available since 2004 and allows to find the best combination of flags iteratively using *exhaustive*, *random*, *one of* and *all but one* search methods.

Machine-learning has been also investigated to predict good s transformations and improve hand-tuned compiler heuristics [27, 35, 34, 9, 40, 6]. These works use genetic programming, supervised learning, decision trees, predictive modeling and other similar techniques to tune compiler heuristics usually for one or a few specific transformations.

In our research, we investigate practical aspects of iterative optimizations such as fine-grain tuning of large applications [18, 15], predicting when to stop iterative search [19], run-time optimizations and program low-overhead adaptation at fine-grain level (procedure or loop) for different behaviors [17], investigating the influence of different datasets on iterative search and program performance [16], using machine learning to speed-up optimization process [6] or to speed-up performance prediction for the effective architecture design space exploration [8].

Many interesting transformation and optimization tools have been developed for the purpose of iterative and adaptive compilation. However, most of them are incompatible with each other and not easily portable across architectures. Moreover, using source-to-source optimizations and pragmas can result in heavy, unreadable and non-portable programs that can perform worse on new architectures, so that additional de-optimization/re-optimization techniques may be needed [23]. What makes things worse is an additional, often unpredictable and unquantifiable interference of the tools with hidden/black-box internal compiler optimizations. The most important motivation for our proposed framework is that current tools tend to rewrite and duplicate parts that are currently available inside most compilers, simply because compilers themselves are often seen as untouchable: they rely on intricate transformation engines, undocumented and multi-purpose heuristics, pass orderings and intermediate representations fragile to any modification, providing no support for external tuning and on-demand application of specific transformations. We would like to discrown these myths in this paper and show that current compilers can be used as powerful, flexible yet stable iterative interaction transformation toolsets, their existing heuristics being initially treated as black-boxes, i.e. the inputs and outputs are known but the internal behavior is not, and progressively learned to adapt to a give program on a given architecture.

We developed an Interactive Compilation Interface (ICI) for the commercial PathScale EKOPath Compiler to bias *all* internal optimization decisions and their parameters externally. It is a non-intrusive, stable and flexible way to tune programs at a function, loop or instruction level for best performance, code size, power consumption and any other objective function supported by existing heuristics and an external driver. Current version of the ICI works in the informative and reactive mode, when external tools rather than querying a compiler to apply some specific transformation on a given part of the program, first obtain information from an interactive compiler about all possible legal transformations and their parameters for a specific part of the code and later respond to the compiler to either keep compiler decisions or change them based on statistical and machine learning techniques. This can considerably reduce optimization search space since there is no need to attempt to traverse through illegal or not supported transformations. Still, reactive nature of our method allows external tools to select and parameterize any possible legal sequences of transformations.

We extend the current ICI to support GCC, whose recent versions feature a large number of advanced transformations but with ineffective heuristics. We plan to add support for the reordering of the optimization passes in the GCC to our ICI on a function or loop level, since it already has a relatively clean description of such passes on a global program level. We suggest to substitute current multiple transformation tools with such iterative interactive compilers and use external tools to fine-tune their optimization heuristics. Tools that achieve best results can later be easily and transparently added to the compiler, all the users being immediately able to take advantage of this improvement. Moreover, using unified ICI allows optimization knowledge reuse among different programs and architectures with statistical and machine learning techniques. It also simplifies future compiler development when adding new transformations: e.g., their optimization heuristic can be automatically and continuously learned with machine-learning techniques in the external driver.

2 Motivation

To motivate our research on an open iterative research compiler, we decided to consider some current optimization techniques for `mgrid` application from SPEC CPU2000FP benchmark suite [33]. From [18] we know that source-to-source loop tiling(blocking) and unrolling on `mgrid` from SPEC CPU95FP can reduce its execution time. We used the same source-to-source transformation tool from [18] and hill-climbing search to iteratively find best tiling and unrolling factors for two most-time consuming loops from procedures *resid* and *psinv* for this benchmark on a recent AMD Athlon 64 3700+ plat-

Program version:	Loop from procedure/ transformation:	source-to-source transformation factor:	internal transformation factor:	speedup:
Best variant found with source-to-source transformer	resid/loop tiling	not-found	15 and 182	1.13
	resid/loop unrolling	8	2	
	psinv/loop tiling	not-found	18 and 204	
	psinv/loop unrolling	8	2	
Best variant found with interactive iterative compiler	resid/loop tiling	not-needed	60	1.17
	resid/loop unrolling	not-needed	13	
	psinv/loop tiling	not-needed	9	
	psinv/loop unrolling	not-needed	14	

Table 1. Comparison of best factors found for mgrid benchmark when using source-to-source transformation tool and interactive iterative compiler

form. Later, we applied the same hill-climbing search but using our iterative interactive PathScale EKOPath Compiler.

The results presented in table 1 show that though we reduced execution time using our older source-to-source transformer but interference with internal compiler transformations diminished the result. Previously, we often had to reduce the optimization level of the compiler to remove such ambiguities and sometimes could even improve results, but supporting less transformations than the compiler we could miss some important optimizations. Using interactive iterative compiler, we both avoid this problem and obtain much better result. Moreover, performing optimizations only inside compiler, we reduce and simplify the optimization space since compiler suggests only legal transformations. In addition, many transformations currently applied by the compiler are not profitable (as noticed in [36]) which can also improve the precision of machine learning techniques that we currently use to improve compiler heuristics, quickly find best optimizations or predict best performance, for example (extension of [6, 8]).

Since we bias compiler optimization decision instead of querying it to apply some specific transformation at a particular place in the program, we naturally force compiler to apply aggressively all possible transformations and later de-select unnecessary transformations or change parameters to apply sequences of transformations. Similar method has been used in PathOpt optimization tool (*all but one* search strategy) and later in [29, 30]. The optimization target in these tools are global/procedure-level compiler flags and the main goal is to speed-up the search. However, we noticed, that when optimizing program at fine-grain level in a complex optimization space, turning on all optimizations and later de-selecting some of them would not speed-up the search since multiple ambiguous interactions of various transformations could often considerably degrade the performance. Hence, we use this method to naturally bias compiler optimization decisions externally and we use machine learning techniques to find best optimizations in large optimization spaces quickly (as in [6]) and run-time versioning to further speed-up the search and to adapt to different program behaviors at run-time (as in [17]). We should also note, that some of the source-to-source automatic or manual transformations are still needed since they may be syntactic and difficult to implement inside a compiler. We are currently implementing an ICI for an open-source GCC and

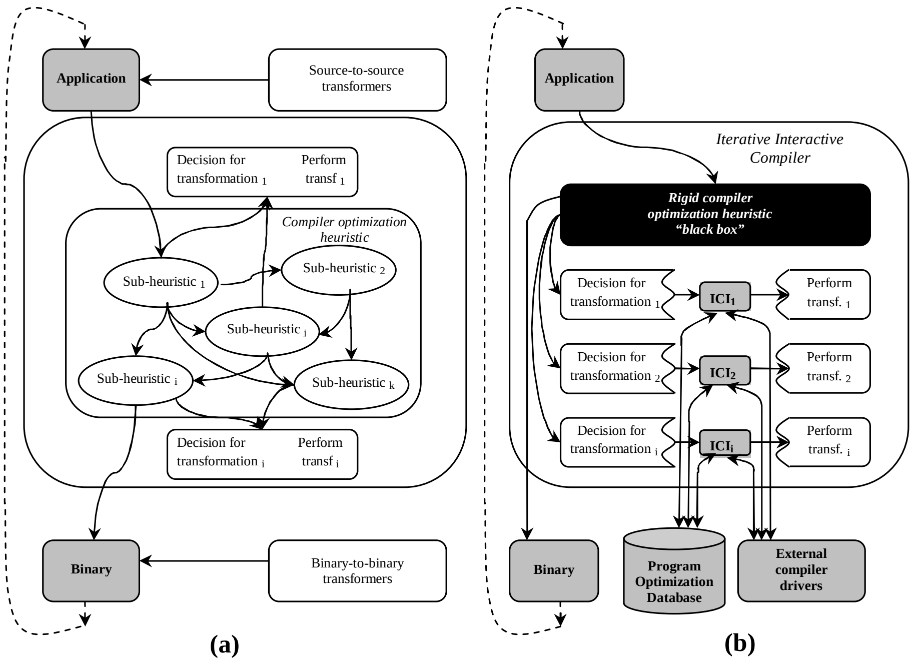

Application
Source-to-source transformers
Decision for transformation 1
Perform transf 1
Compiler optimization heuristic
Sub-heuristic 1
Sub-heuristic 2
Sub-heuristic j
Sub-heuristic i
Sub-heuristic k
Decision for transformation i
Perform transf i
Binary
Binary-to-binary transformers
(a)
Application
Iterative Interactive Compiler
Rigid compiler optimization heuristic "black box"
Decision for transformation 1
ICI1
Perform transf. 1
Decision for transformation 2
ICI2
Perform transf. 2
Decision for transformation i
ICIi
Perform transf. i
Binary
Program Optimization Database
External compiler drivers
(b)

Fig. 1. Internals of (a) current compilers and (b) interactive compilers

plan to gradually add more transformations to this compiler while learning their heuristics automatically.

3 Compiler Framework

Based on our previous experience on iterative optimizations [18, 15, 17, 6, 16], the practical open iterative interactive compiler should have the following features:

- allows simple and unified mechanism to obtain information about all compiler decisions externally and bias them;
- reuses all the compiler program analysis routines to avoid duplications in external optimization tools;
- transparent to user - no project modifications needed;
- removes unnecessary interactions between source-to-source optimizers, compiler and back-end binary-to-binary translators;
- narrows down the optimization search space by using only legal transformations for a given application;
- allows fine-grain (function, loop or instruction level) tuning to get better quality code;
- simplifies compiler development and tuning for new architectures;
- allows reuse of information among different programs and architectures;
- allows modular pluggable third-party transformations and optimization tools.

To address these issues, we suggest the structure of a practical iterative compiler with an Interactive Compilation Interface as shown in Figure 1. This figure depicts an abstract representation of current compilers with hardwired and often ineffective optimization heuristics (Figure 1a) and of the suggested interactive compiler (Figure 1b)). Whenever compiler optimization heuristic makes a potentially ineffective decision to apply some transformation, compiler has to "push out" all the analysis information preceded this decision and provide a user an ability to modify this decision and parameters of this transformation externally through an ICI. We can still treat compiler heuristic as a black box, i.e. where only inputs (compiler decisions) and desired output (performance metrics or other objective function) are known without knowledge about internal structure and transformation interactions, but exposing all decisions at all possible levels (global, function, loop, instruction) allows external tools to automatically learn this behavior and adapt to specific programs and architectures.

The pitfall would be to reveal the compiler intermediate representation and libraries, to a point where it would rigidify the whole internals and stiffen further evolution. To avoid this pitfall, we choose to control the *decision* process itself, granting access the only *high-level features* needed to effectively take a decision. This restriction is compatible with our fine-tuning and fine-grained interaction, and allows to tune programs for best performance, code size, power consumption; we also believe it allows for joint architecture-compiler design-space exploration. By exposing only the decisions that arise from the opportunities suggested by the program syntax and semantics (e.g., detecting that two loops are candidates for tiling), and only when the associated legality checks are satisfied (e.g., checking dependence properties), we dramatically reduce the combinatorial space of program transformation sequences that is searched by external optimization drivers. In fact, we only provide the external optimizer with the combinations currently suggested by the opportunity and legality analyses *triggered on the compilation unit*, granting it access to the only program features embedded into the compiler's specific optimization passes. We believe that this approach of interacting with a compiler will simplify the tuning process of new optimization heuristics and will eventually simplify the whole compiler design where compiler heuristics will be learned automatically, continuously and transparently for a user using statistical and machine learning techniques.

We are developing several communication methods with an interactive compiler - through external file/database, as a client/server connection and through internal calls with a tightly coupled external tool. As a first step, we decided to use external file tor communicate with a compiler similar to common feedback-directed compilation as shown in figure 2. In a *write* mode simply invoked by setting environment variable PATHSCALE_ICI_W to 1, compiler generates an external XML transformation file that contains information about all applied transformations, their parameters and available analysis information. This communication method allows transparent optimizations without any modifications of the source code or project files. An external tool can parse this file with any standard XML parser and modify parameters of existing transformations or disable them. This file is later feeded back to the compiler in the *read* mode by setting environment variable PATHSCALE_ICI_R to 1. In this mode compiler reads and parses modified XML transformation file while optimizing program

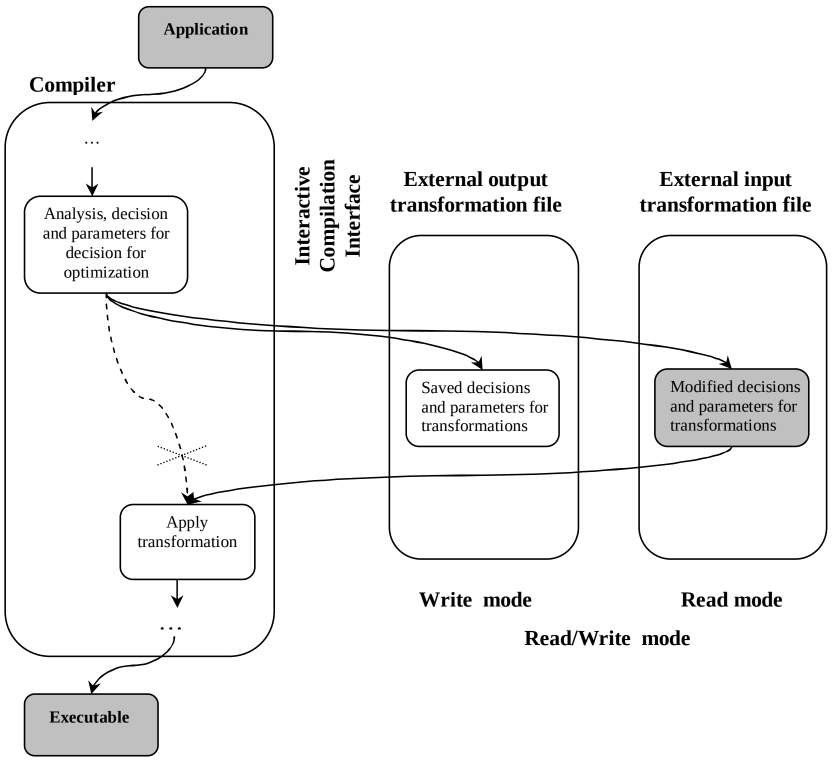

Application
Compiler
...
Analysis, decision and parameters for decision for optimization
Apply transformation
...
Executable
Interactive Compilation Interface
External output transformation file
External input transformation file
Saved decisions and parameters for transformations
Modified decisions and parameters for transformations
Write mode
Read mode
Read/Write mode

Fig. 2. Communication with external tools through transformation file

and substitutes its heuristic decisions and parameters with the matched ones from the transformation file. A sample transformation output for `swim` is shown in figure 3.

When applying transformations that may change loop ordering such as loop interchange, fusion/fission, tiling and others, the subsequent optimization decisions of the compiler can change and will not be matching with the optimization order in the external transformation file. This may result in skipping some externally modified decisions that can cause inconsistencies for external tools when automatically learning the behavior of the program. In such cases, iterative recompilation is required, when interactive compiler and external tool iteratively process the transformation file, refine optimization decisions occurred at each iteration and recompile the program until the the desired sequence of decisions is achieved. To enable such recompilation a *read/write* mode of the interactive compiler is used (when both environment variables PATHSCALE_ICI_W and PATHSCALE_ICI_R are set to 1). In such mode, compiler reads and matches the transformation file with internal optimization decisions, and at the same time produces a new transformation file that contains both modified and unmatched optimization decisions. The iterative recompilation algorithm is shown in figure 4. Naturally, this mode is also used to apply any legal sequences of transformations, thus demonstrating how a

```
<?xml version="1.0"?>
<compiler_ici>
 <file_name="swim.f">

  <transformation name="unroll_and_peel">
   <function>calc1</function>
   <loop_number>4</loop_number>
   <depth>1</depth>
   <decision>4</decision>
   <factor>7</factor>
  </transformation>

  <transformation name="unroll_and_peel">
   <function>calc1</function>
   <loop_number>3</loop_number>
   <depth>1</depth>
   <decision>4</decision>
   <factor>7</factor>
  </transformation>
  …

</file_name>
</compiler_ici>
```

Fig. 3. Example of the transformation XML file for `swim`

```
    clear transformation_file_out.xml
    set PATHSCALE_ICI_W to 1
    compile program
        (write transformation_file_out.xml)
    set PATHSCALE_ICI_R to 1
_label_recompile:
    copy transformation_file_out.xml to
        transformation_file_in.xml
    modify transformation_file_in.xml if needed
    compile program
        (read transformation_file_in.xml,
        write transformation_file_out.xml)
    if transformation_file_in.xml not the same
        as transformation_file_out.xml
        go to _label_recompile
```

Fig. 4. Iterative recompilation algorithm to apply sequences of transformations

compiler with a hardwired heuristic can become a flexible transformation tool with our Interactive Compilation Interface. However, for some large applications, using external file and *read/write* compiler mode for interaction with external tools may require several recompilation and can be time consuming. This motivated us to develop a prototype of a client/server communication method where decisions can be modified during during compilation time and therefore no further recompilation is needed.

Currently, we added support to modify internal PathScale EKOPath compiler optimization decisions for the following transformations:
inlining, array padding (global/local), loop fusion/fission,
loop interchange, loop blocking, loop unrolling, register tiling,
prefetching.

4 Tools and Experiments

We expect to substitute multiple often non-portable and non-compatible transformation tools with our iterative interactive compiler as shown in figure 5. In this case, optimizations will be performed continuously and transparently to user, i.e. it will not require any modifications of a source code or project files. All information about best found optimizations on a given architecture is saved in a *Program Transformation Database* kept along with a program. This simplify application development, optimization and portability since no information about optimizations is now hardwired in the source code of the program and there is no dependence on multiple external tools that may not be available on some architectures. Moreover, *Program Transformation Database* keeps information about all best possible optimizations for different program behaviors on different architectures (as described in [17]). Therefore, whenever program is ported to a new platform, the optimization process can start from already found best configurations from multiple programs thus reusing optimization knowledge among different

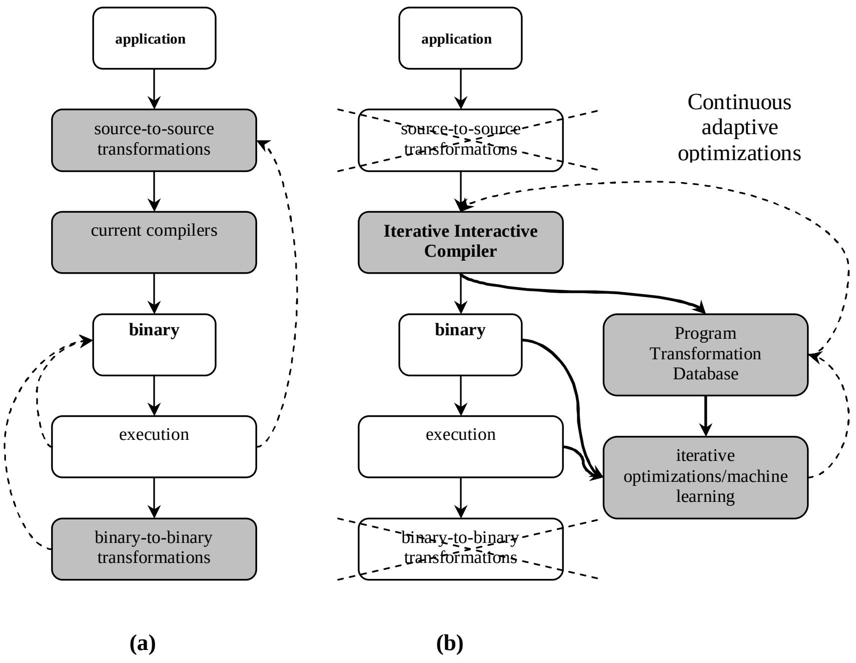

application
source-to-source transformations
current compilers
binary
execution
binary-to-binary transformations
(a)
application
~~source-to-source transformations~~
Continuous adaptive optimizations
Iterative Interactive Compiler
binary
Program Transformation Database
execution
iterative optimizations/machine learning
~~binary-to-binary transformations~~
(b)

Fig. 5. Iterative optimization scenario using iterative interactive compiler

programs and architectures, behaving as a collective compilation system and considerably narrowing down the optimization search space.

Since 2004, we and our colleagues actively used our iterative interactive compiler in different projects and developed or prototyped the following support tools and optimization drivers.

- Continuous iterative optimization driver with run-time adaptation at function, loop-level or instruction level using low-overhead phase detection technique (as described in [17]. We use *exhaustive*, *random* and *hill-climbing* search strategies (as in [7, 18, 15]). We also use a *all but one* search strategy on a fine-grain level similar to the one implemented in the PathOpt tool from the original PathScale EKOPath compiler distribution where global compiler flags are turned all on at the first step and later turned off one by one.
- Driver to continuously collect all possible optimization parameters. This driver is useful when compiler optimization heuristics is treated as a black box and its behavior is learned automatically to collect all varieties of optimization decisions and parameters automatically and transparently to a user, instead of listing them separately and keeping them up-to-date.
- Driver to automatically and continuously rebuild compiler optimization heuristic, and adapt to a specific architecture using statistical methods and collective optimization knowledge reuse among different programs and architectures.

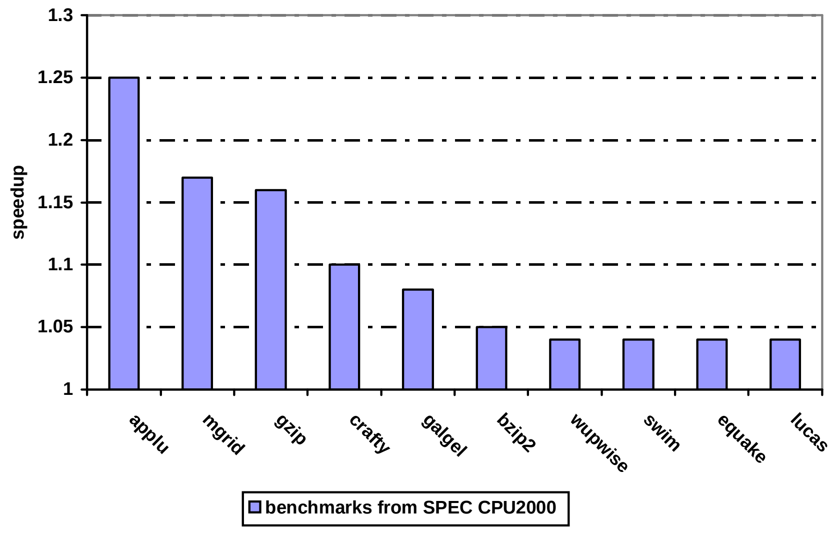


Fig. 6. Speedups of several SPEC CPU2000 applications in comparison with -Ofast optimization level when using interactive PathScale EKOPath compiler with hill-climbing search

– Prototype framework to replace a model-based compiler heuristic with automatically learned one. We use our iterative interactive compiler together with the WEKA tool [3], which is an open-source machine learning software package. Our preliminary results target loop interchange; we will revisit the main optimizations for which machine learning techniques have been proposed so far, as well as experiment with more challenging ones.

We developed multiple support tools and external optimization drivers for our iterative interactive compiler. We already created various tools to support our ICI-enabled compiler and we developed several external optimization tools that uses iterative search to find best transformations and their parameters to minimize execution time of programs. We use *exhaustive*, *random* and *hill-climbing* search (as in [7, 18, 15]), *all but one of* search (implemented in the PathOpt tool from the original PathScale EKOPath compiler suite [2] when all compiler flags are turned on and later turned-off one by one). We also use this interactive compiler with program optimizations at fine-grain level for run-time program adaptation described in [17].

Since the purpose of this article is mainly to describe the building of an interactive iterative compiler, we decided to leave complex iterative optimization schemes for the journal version of the paper and selected a relatively simple hill-climbing optimization scheme as described in [18, 15]. We performed all experiments on AMD Athlon 64 3700+ at 2.4GHz, with an L1 cache of 64KB and an L2 cache of 1MB, and 3GB of memory; the O/S is Mandriva Linux 2006. We instrumented the open-source commercial PathScale EKOPath Compiler 2.x [2] to enable Interactive Compilation Interface to allow external tuning of its optimization heuristic. It has a mature but ambiguous optimizer with many transformations available, based on the ORC compiler, and is specifically tuned to AMD processors. We selected several programs from the SPEC 2000 suite [33] and ran our search tool continuously and transparently to user until all transformations and their parameters have been analyzed. Whenever possible, we used run-time versioning scheme from [17] to considerably speed-up iterative search and allow further run-time adaptation. We needed from around 500 to 5000 runs (with 16 versions of examined functions during one run) per program to finish optimizations. The speedups shown in figure 6 in comparison with the best -Ofast optimization level of the EKOPath compiler demonstrate that it is possible to beat the state-of-the-art compiler even on large programs with the most aggressive optimization level enabled using simple Interactive Compilation Interface and external iterative optimization drivers. We will describe all other optimization scenarios in more detail in the journal version of the paper and will make all the software, source codes and data publicly available at [4].

5 Conclusions and Future Work

In this article we demonstrated a simple, practical and non-intrusive way to turn current rigid compilers into powerful interactive transformation toolset with an Interactive Compilation Interface that allows to bias compiler optimization decisions externally. We show how to avoid the pitfalls of rigidifying the compiler internals, while granting access to rich-enough features to take performance-critical decisions. We assist the external optimization tools in considerably reducing the size of the optimization search space by analyzing only possible transformations, and in continuously collecting the most interesting sets of transformation parameters. We developed an ICI for the commercial open-source PathScale EKOPath Compiler and, within last 2 years, developed different support tools to optimize programs at loop or instruction level continuously and transparently to a user. We use it to automatically optimize programs for the best performance, code size, power consumption and hardware designs. We plan to make all the software publicly available at [4].

Based on this work, we are currently developing a unified extensible and portable ICI in the latest version of GCC [5] with a support from IBM, Philips (NXP), STMicro, ARC and multiple universities within HiPEAC network of excellence [1]. We enable an access to the most influential compiler transformations (including OpenMP directives) with ineffective optimization heuristics and enable optimization pass reordering at a function or loop level. We are working on the optimization naming conventions to enable portability and automatic knowledge reuse using machine learning between different compilers and their versions. We plan to add ICI to the JIT-compilers (Jikes, .NET compilers) to unify the run-time optimizations as well. One of the most advantages of a unified ICI is that it enables life-long, whole-program compilation research with collective reuse of the knowledge (program features, analysis results and transformation decisions) across different programs and architectures without the overhead of breaking-up the compiler into a set of well-defined compilation components (communicating through persistent intermediate languages), even if such an evolution could be desirable at some point (but much more intrusive).

We are using our toolset in the EU-funded MilePost, SARC and GGCC projects. Within MilePost, we aim at dramatically changing and simplifying the design of future compilers on rapidly evolving hardware by automatically and continuously learning the best optimization settings for a given program, context, platform and any given set of compiler transformations. Within SARC, we facilitate collective optimization-space exploration of the architecture and compiler, on a heterogeneous chip multi-processor. Within GGCC, we contribute to the emergence of a production-quality standard for whole-program analysis and optimization. We believe this is a major research and development direction towards a practical and general-purpose development toolset based on integrative compilation.

6 Acknowledgments

The authors would like to thank Olivier Temam and Michael O'Boyle for the helpful discussions and support throughout the project, and anonymous reviewers for a valuable feedback. This research is partly funded by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC).

References

1. European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). `http://www.hipeac.net`.
2. PathScale EKOPath Compilers. `http://www.pathscale.com`.
3. WEKA: Open-source machine learning software. `http://www.cs.waikato.ac.nz/ml/weka`.
4. Interactive Compilation Interface for PathScale EKOPath Compiler. `http://sourceforge.net/projects/pathscale-ici`, 2004.
5. GCC Interactive Compilation Interface. `http://sourceforge.net/projects/gcc-ici`, 2006.
6. F. Agakov, E. Bonilla, J.Cavazos, B.Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2006.
7. F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *Proceedings of the Workshop on Profile and Feedback Directed Compilation*, 1998.
8. J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M. O'Boyle, G. Fursin, and O. Temam. Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the International Conference on Compilers, Architecture, And Synthesis For Embedded Systems (CASES 2006)*, October 2006.
9. J. Cavazos and J. Moss. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004.
10. K. Cooper, A. Grosul, T. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. ACME: adaptive compilation made efficient. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.
11. K. Cooper, P. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–9, 1999.
12. K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1), 2002.
13. A. Epshteyn, M. Garzaran, G. DeJong, D. Padua, G. Ren, X. Li, K. Yotov, and K. Pingali. Analytic models and empirical search: A hybrid approach to code optimization. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Hawthorne, NY, USA, 2005.

14. B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2005.
15. G. Fursin. *Iterative Compilation and Performance Prediction for Numerical Applications*. PhD thesis, University of Edinburgh, United Kingdom, 2004.
16. G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam. Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2007) (to appear)*, January 2007.
17. G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *Proceedings of the International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2005)*, pages 29–46, November 2005.
18. G. Fursin, M. O'Boyle, and P. Knijnenburg. Evaluating iterative compilation. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computers (LCPC)*, pages 305–315, 2002.
19. G. Fursin, M. O'Boyle, O. Temam, and G. Watts. Fast and accurate method for determining a lower bound on execution time. *Concurrency: Practice and Experience*, 16(2-3):271–292, 2004.
20. M. Haneda, P. Knijnenburg, and H. Wijshoff. Generating new general compiler optimization settings. In *Proceedings of the 19th annual international conference on Supercomputing (ICS'05)*, pages 161–168, New York, NY, USA, 2005.
21. M. Haneda, P. Knijnenburg, and H. Wijshoff. On the impact of data input sets on statistical compiler tuning. In *Proceedings of the Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL)*, 2006.
22. K. Heydemann and F. Bodin. Iterative compilation for two antagonistic criteria: Application to code size and performance. In *Proceedings of the 4th Workshop on Optimizations for DSP and Embedded Systems, colocated with CGO*, 2006.
23. S. Hines, P. Kulkarni, D. Whalley, and J. Davidson. Using de-optimization to re-optimize code. In *Proceedings of the EMSOFT Conference*, pages 114–123, 2005.
24. T. Kisuki, P. Knijnenburg, M. O'Boyle, and H. Wijshoff. Iterative compilation in program optimization. In *Proceedings of the Workshop on Compilers for Parallel Computers (CPC2000)*, pages 35–44, 2000.
25. P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. In *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 12–23, 2003.
26. F. Matteo and S. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.
27. A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *Proceedings of the International Conference on Artificial Intelligence: Methodology, Systems, Applications*, LNCS 2443, pages 41–50, 2002.
28. Z. Pan and R. Eigenmann. Rating compiler optimizations for automatic performance tuning. In *Proceedings of the International Conference on Supercomputing*, 2004.
29. Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 319–332, 2006.
30. Z. Pan and R. Eigenmann. Fast automatic procedure-level performance tuning. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT)*, Seattle, WA, September 2006. IEEE Computer Society.

31. R. Pinkers, P. Knijnenburg, M. Haneda, and H. Wijshoff. Statistical selection of compiler options. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 494–501, 2004.
32. B. Singer and M. Veloso. Learning to predict performance from formula modeling and training data. In *Proceedings of the Conference on Machine Learning*, 2000.
33. The Standard Performance Evaluation Corporation. `http://www.specbench.org`.
34. M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, pages 123–134, 2005.
35. M. Stephenson, M. Martin, and U. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 77–90, 2003.
36. S. Triantafyllis, M. Vachharajani, and D. August. Compiler optimization-space exploration. In *Journal of Instruction-level Parallelism*, 2005.
37. S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 204–215, 2003.
38. R. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proc. Alliance*, 1998.
39. K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 63–76, 2003.
40. M. Zhao, B. R. Childers, and M. L. Soffa. A model-based framework: an approach for profit-driven optimization. In *Third Annual IEEE/ACM Interational Conference on Code Generation and Optimization*, pages 317–327, 2005.

Track: Computer Architecture

Feasability of Combined Area and Performance Optimization for Superscalar Processors Using Searching

Sven van Haastregt[1] and Peter M.W. Knijnenburg[2]

[1] LIACS, Leiden University, The Netherlands
svhaastr@liacs.nl
[2] CSA, Informatics Institute
University of Amsterdam, The Netherlands
peterk@science.uva.nl

Abstract. When designing embedded systems, one needs to make decisions concerning the different components that will be included in a microprocessor. An important issue is the chip area vs. performance trade-off. In this paper we investigate the relationship between chip area and performance for superscalar microprocessors. We investigate the feasability to obtain a suitable configuration by searching. We show that our approach gives a good configuration after 100 to 150 iterations using a simple random search algorithm. This shows the feasibility of our approach, in particular when more sophisticated search algorithms are employed as we plan in future work.

1 Introduction

Current embedded systems require high performance. Therefore, several current and many future embedded processors are out-of-order or even simultaneous multithreaded [9]. A drawback of these types of processor is that they consume much silicon area because of the complicated control structures required to support out-of-order execution [7]. This may be problematic for embedded systems where silicon area is expensive. Therefore, it is important to tune the architecture in such a way that maximum performance is achieved using a minimal amount of resources. Obviously, this is very difficult for general purpose processors, but in the case of embedded processors that only run a limited set applications, it may be possible to select a restricted set of resources in such a way that high performance still is achieved. For in-order processors there exist many approaches to explore the design space [5]. For example, PICO is an automatic system to explore application specific VLIW processors [8]. In the Artemis project [13] two different frameworks for the simulation phase are adopted: Spade [10] provides a model for rapid high level architecture performance simulations, and Sesame [14], provides a method for evaluating designs at multiple abstraction levels.

Recently, Eyerman et al. have proposed methods to explore the design space of out-of-order processors [4] focussing mainly on energy. In this paper, we study the feasability to automatically search for good out-of-order processors configurations for specific applications, paying attention to both performance and area. That is, we want to find a processor configuration that is small but powerful enough for specific applications.

We use the SimpleScalar toolset as the design space to explore [1]. SimpleScalar allows us to set many architectural parameters. Each component has a different effect on the final performance. Furthermore, there exist dependencies between the several components. For example, increasing the number of arithmetical units will not increase performance, unless multiple instructions can be executed in parallel. It requires quite some analysis to find all dependencies between the various components and the list of dependencies quickly becomes complex. As we show in Section 2, even with a relatively small amount of possible design options (from now on referred to as *tuning parameters*), the search space is huge. Therefore, we employ a random search algorithm to explore only a fraction of this space. We show that in this way, using only about 100 to 150 configurations, we can find a high performance architecture that is much smaller than a general purpose architecture. This shows the feasibility of our approach, particularly when more sophisticated search algorithms would be developed as we plan in future work.

This paper is structured as follows. In Section 2, we describe the experiments we have performed with this new approach, and Section 3 contains the results of these experiments and we also give a short discussion. In Section 4, we mention some possible directions for future work. Section 5 summarizes this paper.

2 Experimental Setup

In this section, we discuss how we generate configurations, how performance is measured, the parameters of our experiments and the area model that is being used.

The search algorithm we use in our experiments is the most basic one available: we randomly generate a set of 1000 configurations (without duplicates) using different tuning parameters and then measure the performance and calculate the area of each configuration.

To evaluate the performance of each configuration, we use the *SimpleScalar Tool Set* [1]. The SimpleScalar simulator supports several instruction set architectures. We use the PISA architecture.

We use two applications for our experiments, `ijpeg` and `mpeg2dec`. Both of these programs rely heavily on integer calculations and scarcely on floating point operations. Therefore, we keep the number of floating point arithmetical units constant throughout the experiments. The ijpeg-simulation accounts for a total of about 1.1×10^9 instructions. The mpeg2dec-simulation results in about 1.3×10^8 instructions.

We have selected the following tuning parameters. In an iteration a value from the matching parameter value set is assigned to each parameter.

- **Data cache size**: number of bytes of the first level direct mapped data cache, blocksize of 32 bytes. We use 6 values: { `1024, 2048, 4096, 8192, 16384, 32768` }
- **Instruction cache size**: number of bytes of the direct mapped instruction cache, blocksize of 32 bytes. We use 6 values: { `1024, 2048, 4096, 8192, 16384, 32768` }
- **GShare branch predictor size**: A GShare branch predictor consists of a *w* bits wide shift register (the global history register, containing the history of the *w* most

recently executed branches) and a table containing 2^w bimodal counters [11]. We use 5 values: { `512, 1024, 2048, 4096, 8192` }

– **Branch Target Buffer (BTB) size**: the maximum number of entries in the BTB. We use 6 values: { `1, 64, 128, 256, 512, 1024` }
– **Register Update Unit (RUU) size**: the number of slots available in the RUU, the unit that controls the out-of-order execution. We use 7 values: { `2, 4, 8, 16, 32, 64, 128` }
– **Number of integer ALUs**: the number of integer Arithmetic Logic Units available. We use 5 values: { `1, 2, 3, 4, 5` }
– **Number of memory ports**: the number of ports available to the CPU to access the first level cache. We use 4 values: { `1, 2, 3, 4` }
– **Instruction fetch queue size**: the maximum number of instructions that can be stored in the fetch queue. We use 5 values: { `1, 2, 4, 8, 16` }
– **Instruction issue width**: the maximum number of instructions that can be issued per cycle. We use 3 values: { `2, 4, 8` }
– **Load/Store Queue (LSQ) size**: The LSQ handles the actual memory communication and contains a mechanism that avoids data hazards. We use 4 values: { `2, 4, 8, 16` }

All other possible architecture parameters remain constant throughout the experiment and are set at the SimpleScalar default values. With this set of parameters, more than nine million different configurations are possible.

To obtain an estimate of the area of a particular processor configuration, we use a slightly extended version of the model proposed by Steinhaus et al. [15]. This model provides an area estimate for a superscalar microprocessor design, specified using a SimpleScalar configuration, using analytical and empirical models. Chip area is expressed in λ^2 in order to get a quantity that is independent of the technology used to manufacture the microprocessor. Here, λ is defined as half of the minimum *feature size* which is the size of the smallest transistor, interconnect, etc. that can be produced by using a certain manufacturing process.

3 Results

In this section, we first show performance versus area for 1000 randomly picked parameter settings for two benchmarks, namely ijpeg and mpeg2dec. Next, we show how much performance we obtain when we have area constraints.

3.1 Simulation Results

After running 1000 performance simulations for `ijpeg` and `mpeg2dec`, we produced the plots in Figures 1 and 2. The *x*-axis represents the area correspronding to a single configuration and the *y*-axis shows the performance, which is calculated by:

$$\text{performance} = \frac{1}{\text{number of cycles}}$$

We have normalized the results to the SimpleScalar default configuration, given in Table 2. We also executed four additional simulations for each benchmark, which are plotted using horizontal lines. First, we determined the performance for the minimum and maximum configuration, by selecting the smallest and largest values, respectively, for each tuning parameter in our search space discussed in section 2. These are called "reachable minimum" and "reachable maximum", respectively, in Figures 1 and 2. Next, we determined the absolute lower bound allowed by SimpleScalar by selecting the minimum value for each tuning parameter allowed by the SimpleScalar simulator. Finally, we determined an estimate of the upper bound by selecting very large values for each tuning parameter as listed in Table 1. The sizes of these configurations are listed in Table 3.

Register Update Unit size	2048 slots
Data cache size	16 Megabytes
Instruction cache size	16 Megabytes
GShare branch predictor size	524288 entries
Branch target buffer size	524288 entries
Number of integer ALUs	8 (maximum)
Number of memory ports	8 (maximum)
Instruction issue width	64 instructions per cycle
Instruction fetch queue size	64 instructions
Load/Store Queue size	1024 entries

Table 1: Parameter settings for the configuration that is our estimated upper bound.

Register Update Unit size	16 slots
Data cache size	4 Megabytes
Instruction cache size	16 Megabytes
GShare branch predictor size	2048 entries
Branch target buffer size	512 entries
Number of integer ALUs	4
Number of memory ports	2
Instruction issue width	4 instructions per cycle
Instruction fetch queue size	4 instructions
Load/Store Queue size	8 entries

Table 2: Parameter settings for the SimpleScalar default configuration

Figures1 and 2 show that there is a difference in performance between the minimum reachable and maximum reachable configurations of about a factor of five. Compared to this, the difference between the reachable minimum and the SimpleScalar minimum is quite small. The same applies to the difference between the reachable maximum and

Configuration	Area (Mλ^2)	Speedup	
		`ijpeg`	`mpeg2`
Reachable minimum	11250	1.0	1.0
Reachable maximum	44539	6.4	7.5
Minimal configuration	11168	0.6	0.7
Huge configuration	13764139	7.2	8.5

Table 3: Area and Speedup wrt minimum configuration of reference configurations.

the large SimpleScalar configuration. Thus, the value sets we have chosen for the tuning parameters cover a broad range of the search space.

One immediately notices the four clusters that appear in both plots. These turn out to be caused by the “number of memory ports” parameter: each value for this parameter corresponds to a cluster. Since this parameter has a huge impact on the total area of a configuration, it clearly separates the different classes. This is caused by the amount of additional wiring and logic needed for each memory port. For example, when the number of memory ports is increased by one, the load/store queue requires at least one additional read and write port for each of its SRAM cells. This is because the LSQ must be able to serve an additional read or write operation during a single cycle. The area of several other components, like the register file, TLB and cache, is influenced in a similar manner. However, it seems there is not much to gain anymore when the number of memory ports is higher than 2.

We observe that the majority of the configurations is located below two times the performance of the reachable minimum. However, there are some differences when looking at certain individual configurations. Some have a high performance for the `ijpeg` benchmark while that same configuration does not perform as well as in the `mpeg2dec` simulation, although the performance still lies above the average. Interestingly, this hardly holds conversely: configurations that perform well for the `mpeg2dec` benchmark are also among the best performing configurations of `ijpeg`.

3.2 Improvement under Area Restrictions

In this section, we study how fast the random search algorithm finds a good configuration when we impose a limit on the allowable area. Such a limit is important in practice when a system needs to be fit on a given amount of silicon. The measure of “goodness” we employ in this paper is speedup over the reachable minimum configuration. For a configuration x, $speedup(x)$ is given by:

$$speedup(x) = \frac{performance(x)}{performance(\text{min config.})}$$

Speedups of the reference configurations are shown in Table 3. We use area restrictions ranging from 12,000 to 30,000 Mλ^2. The resulting plots, shown in Figures 3 to 8, are produced by iterating over the set of 1000 configurations. The performance of each

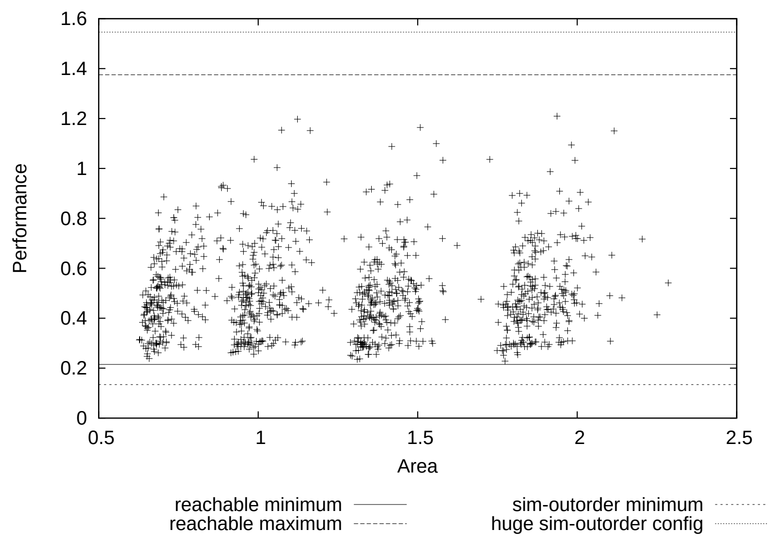


Fig. 1: Area vs. performance ijpeg, normalized wrt default configuration

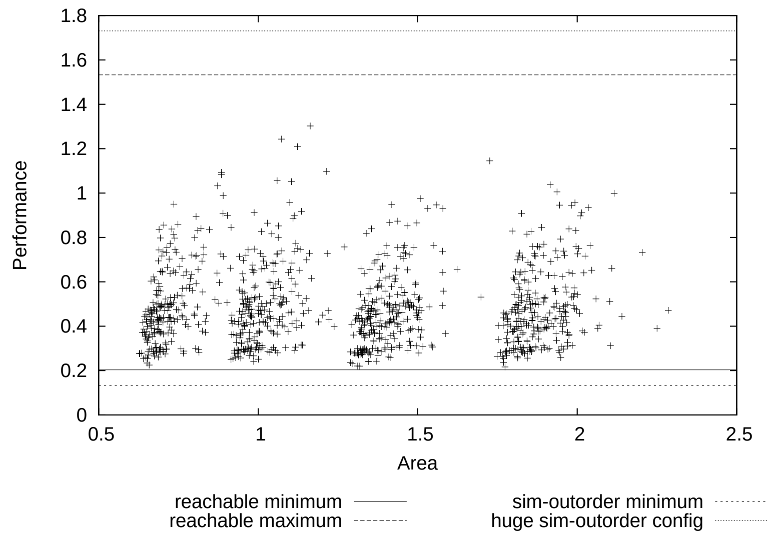


Fig. 2: Area vs. performance mpeg2dec, normalized wrt default configuration

configuration that satisfies the area restriction is plotted. The two different lines in a figure indicate the best configuration encountered so far for both benchmarks.

In Figure 3 we pose a limit that is only slightly larger than the minimum area shown in Table 3. We still produce a configuration that is almost twice as fast as this minimal one. This shows that carefully selecting a few extra resources can be highly effective.

Limits of 13,000 to 15,000 Mλ^2 produce better configurations with speedups of around 4. For `ijpeg` these limits deliver the same configurations, as shown in Table 4. For `mpeg2dec`, a larger value for the limit is used for a larger instruction cache, as shown in Table 4. This indeed gives a higher performance, as shown in Figures 5 and 6.

When the limit allows more than 1 memory port, 2 memory ports are chosen, as shown in Table 3. Figures 7 and 8 show that this gives more performance than 1 port. However, when the limit is 30,000 Mλ^2, 3 ports could be accomodated. However, this value is not chosen, indicating that such extra port does not give extra performance compared to 2 ports.

Finally, we note that we only need around 100 simulations to find good candidates, irrespective of the limit we impose on the area. This shows that our simple approach of using a random search algorithm is already reasonably effective.

Combining the configurations in Table 4 and the speedups from Figures 3 to 8, it is clear that both caches do not need to be that large for the `ijpeg` benchmark. A data cache of 2048 bytes and an instruction cache of 4096 bytes should be sufficient. The reason that data caches can be small lies in the algorithms used in this benchmark: many computations are in essence “local” because discrete transforms are applied to small 8×8 blocks. Also, the compute intensive loops are small so that for `ijpeg` small Icaches can be sufficient. For the `mpeg2dec` benchmark, the same holds for the data cache, but the preferred instruction cache size turns out to be 32 kilobytes. This stresses that one should be careful when evaluating simulation data: the microprocessor configurations that are returned by our approach depend greatly on the benchmark applications used in the simulation step. It shows how important it is to chose the right benchmark suite when designing a microprocessor.

In general, the RUU size needs to be at least 32 and the BTB size at least 64. In the best performing configurations, the branch predictor size varies between the lowest and highest possible values. So it seems this parameter (or the value set we have chosen for it) does not have a big influence on the performance in our experiments. For the `ijpeg` benchmark, the average branch predictor accuracy is about 89%. For the `mpeg2dec` benchmark, the average accuracy is about 97%. In general, the accuracy doesn’t deviate more than 1% from the average for both benchmarks. The minimum number of integer ALUs that need to be included turns out to be three for both benchmark applications. The fetch queue size, issue width and load/store queue size tend to the higher values of the parameter set for a good performance result (≥ 4, ≥ 4, ≥ 8 respectively). The only thing in which both benchmarks significantly differ is the fetch queue size: in general the `mpeg2dec` benchmark performs slightly better when the fetch queue size equals eight or sixteen, compared to configurations that have a smaller fetch queue size.

data cache	instr. cache	branch pred.	BTB	RUU	#ALUs	#memports	FQsize	Issue width	LSQ
ijpeg									
area $\leq$ 12000 Mλ^2									
2048	16384	512	512	16	1	1	2	2	4
area $\leq$ 13000 Mλ^2									
2048	8192	2048	512	32	5	1	4	4	16
area $\leq$ 14000 Mλ^2									
2048	8192	2048	512	32	5	1	4	4	16
area $\leq$ 15000 Mλ^2									
2048	8192	2048	512	32	5	1	4	4	16
area $\leq$ 20000 Mλ^2									
8192	32768	4096	64	64	3	2	8	4	16
area $\leq$ 30000 Mλ^2									
8192	16384	1024	256	128	4	2	4	8	16
mpeg									
area $\leq$ 12000 Mλ^2									
1024	2048	512	512	8	5	1	2	2	16
area $\leq$ 13000 Mλ^2									
2048	8192	2048	512	32	5	1	4	4	16
area $\leq$ 14000 Mλ^2									
2048	32768	1024	128	64	3	1	4	4	8
area $\leq$ 15000 Mλ^2									
2048	32768	1024	128	64	3	1	4	4	8
area $\leq$ 20000 Mλ^2									
8192	32768	4096	64	64	3	2	8	4	16
area $\leq$ 30000 Mλ^2									
2048	32768	2048	256	128	5	2	8	4	16

Table 4: Best Configurations found by random search when applying size constraints

4 Future Work

In this paper, we have used a very simple random search algorithm. The results of the simulations are not used for any feedback. Doing so could improve the search. For example, genetic algorithms can be used. Another direction is by applying data mining techniques on the obtained data, which consists of the confi gurations together with their estimated area and computed performance to create heuristics in order to decrease the size of the search space. An example heuristic can restrict the number of ALUs to the number of instructions that can be fetched simultaneously. Another heuristic can prevent a confi guration from having more LSQ slots than RUU slots. Furthermore, one could try to improve the performance simulation step. A possible way to do this, is to use small, but representative inputs for the benchmark applications used in the simulations [3]. Another approach could use statistical simulation [12,2]. We can also apply Pareto-Front Arithmetics [6] to minimize the part of the design space to be evaluated.

5 Conclusion

In this paper we have demonstrated the feasibility of an iterative approach to the problem of fi nding suitable microprocessor confi gurations: we can fi nd a a high performance confi guration that satisfi es a given area restriction using a simple search algorithm and a limited number of iterations. We have shown that even a small increase in the resources compared to a minimal confi guration can give a speedup of 2.5, which implies that tuning a processor can be highly effective. Our results suggest that around 100 evaluations could be suffi cient. However, this can still be too time consuming, in particular when several applications need to accommodated. Therefore, in future work, we focus on reducing this number by designing more sophisticated search algorithms than the random search from this paper.

References

1. T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.
2. L. Eeckhout and K. De Bosschere. Hybrid analytical-statistical modeling for efficiently exploring architecture and workload design spaces. In *Proc. PACT*, 2001.
3. L. Eeckhout, H. Vandierendonck, and K. de Bosshere. Quantifying the impact of input data sets on program behavior and its applications. *J. of Instruction-Level Parallelism*, 5:1–33, 2003.
4. S. Eyerman, L. Eeckhout, and K. De Bosschere. Efficient design space exploration of high performance embedded out-of-order processors. In *Proc. DATE*, 2006.
5. M. Gries. Methods for evaluating and covering the design space during early design development. *Integration, the VLSI Journal*, 38(2):131–183, 2004.
6. C. Haubelt and J. Teich. Accelerating design space exploration using pareto-front arithmetics. In *Proc. ASP-DAC*, pages 525–531, 2003.
7. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

8. V. Kathail, S. Aditya, R. Schreiber, B.R. Rau, D. Cronquist, and M. Sivaraman. PICO: Automatically designing custom computers. *IEEE Computer*, 35(9):39–47, 2002.
9. M. Levy. Multithreaded technologies disclosed at MPF. *Microprocessor Report*, November 2003.
10. P. Lieverse, P. van der Wolf, E. Deprettere, and K. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. In *Proc. SiPS*, pages 181–190, 1999.
11. S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Lab., 1993.
12. S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *Proc. PACT*, pages 15–24, 2001.
13. A. D. Pimentel, , P. Lieverse, P. van der Wolf, L.O. Herzberger, and E. F. Deprettere. Exploring embedded-systems architectures with artemis. *IEEE Computer*, 34(11):57–63, 2001.
14. A.D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstyraction levels. *IEEE Trans. on Computers*, 55(2):99–112, 2006.
15. M. Steinhaus, R. Kolla, J. Larriba-Pey, T. Ungerer, and M. Valero. Transistor count and chip-space estimation of simplescalar-based microprocessor models. In *Proc. Workshop on Complexity-Effective Design*, 2001.

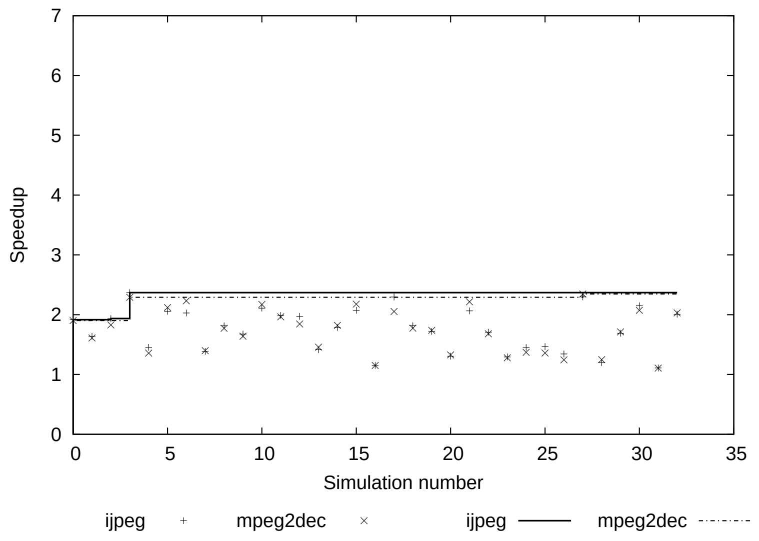


Fig. 3: Area $\leq$ 12000Mλ^2

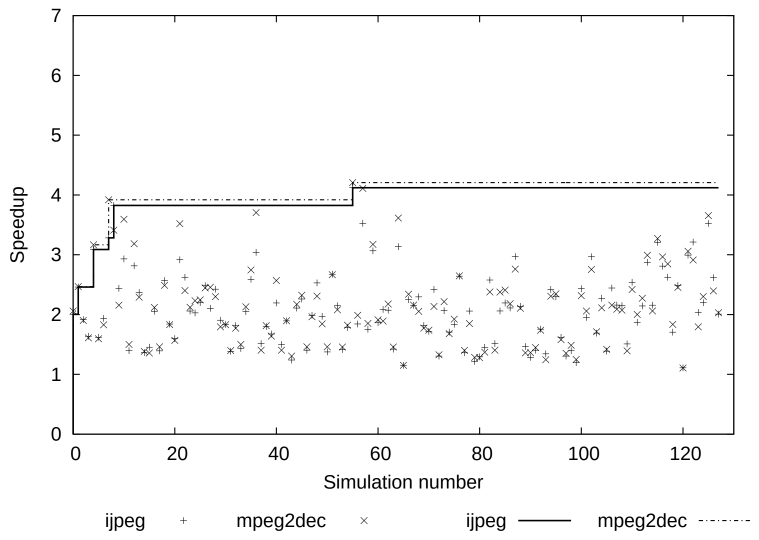


Fig. 4: Area $\leq$ 13000Mλ^2

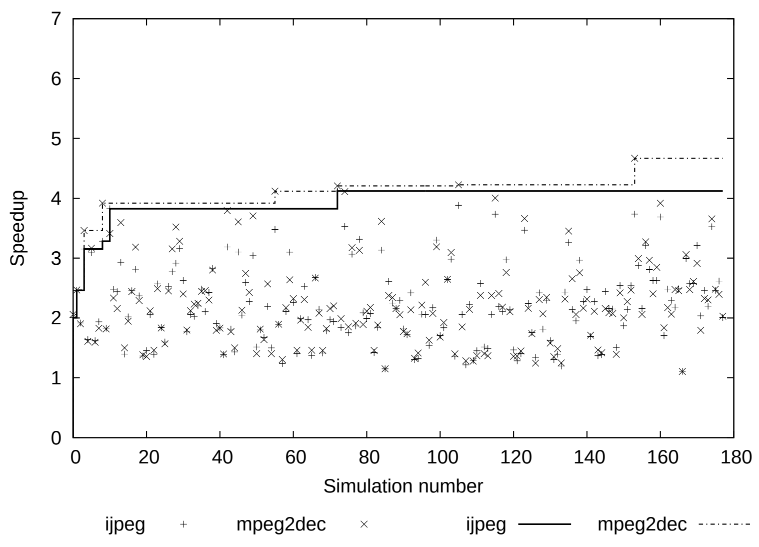


Fig. 5: Area ≤ 14000Mλ^2

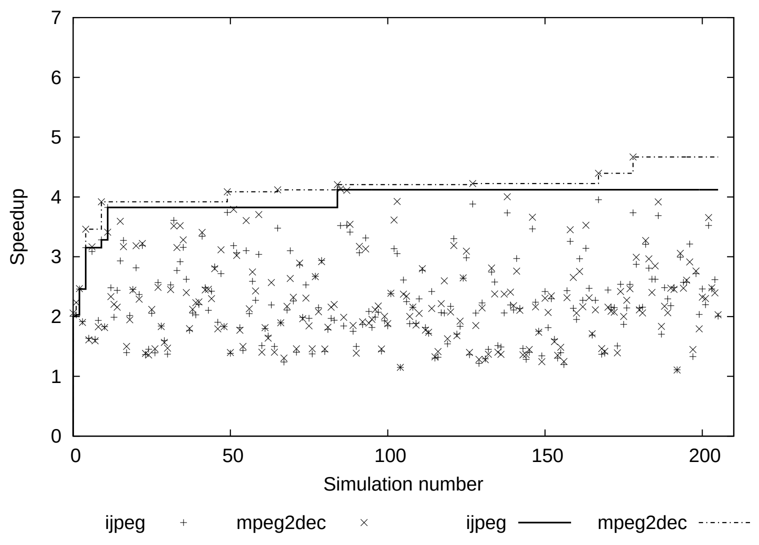


Fig. 6: Area ≤ 15000Mλ^2

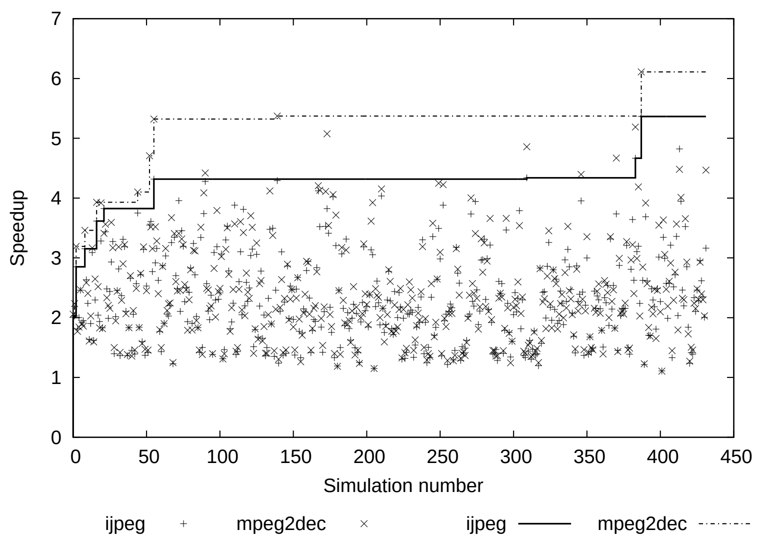


Fig. 7: Area $\leq$ 20000Mλ^2

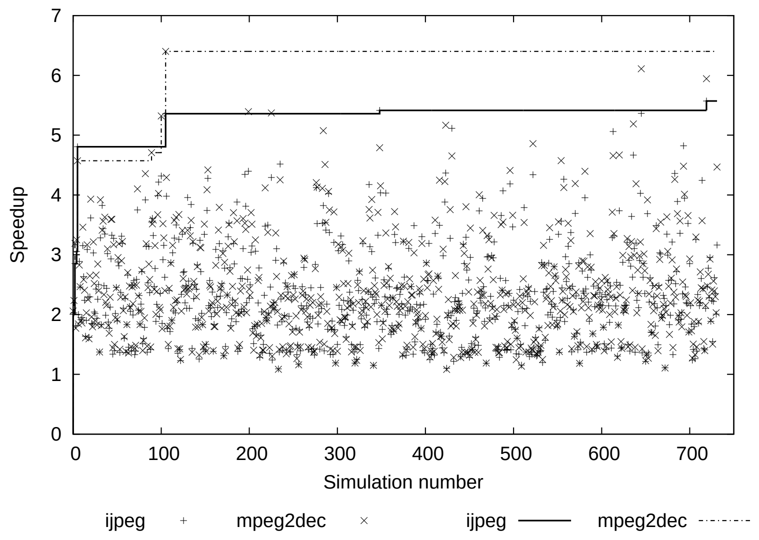


Fig. 8: Area $\leq$ 30000Mλ^2

On the Comparison of Regression Algorithms for Computer Architecture Performance Analysis of Software Applications [*]

ElMoustapha Ould-Ahmed-Vall, James Woodlee, Charles Yount, Kshitij A. Doshi
Intel Corporation
5000 W Chandler Blvd
Chandler, AZ 85226
eouldahm@ece.gatech.edu and {jim.woodlee,chuck.yount,kshitij.a.doshi}@intel.com

Abstract

The ability to provide diagnostic information for workload performance is of great value in the performance tuning process. Not only can it orient the tuning process by identifying key performance issues, it can also be used to estimate the severity of each performance issue and the potential gain from addressing it.

This work investigates the ability of some of the most popular machine learning regression algorithms to provide this diagnostic information. Five regression algorithms are trained using real performance data collected on an Intel® Core™2 Duo processor desktop machine. The algorithms are compared along two axes, prediction quality and usefulness of output, in order to gain key insights into the causes and severity of performance issues.

Although several techniques are found to demonstrate good prediction quality, our study shows that the model-tree-based technique (M5') gives superior interpretability. This class of algorithm produces models that can be used, not only to predict performance, but also to indicate the sources of potential performance improvement and to quantify the potential performance gain. This information can be used to direct performance optimization efforts by prioritizing performance problems.

1 Introduction

Workload performance analysis is used to tune applications to achieve the best possible performance (e.g., shortest execution time) on a given architecture. It can also be used to compare different implementation alternatives during the design and implementation of new applications. Traditionally, performance analysis is conducted by counting the number of occurrences of micro-architectural events, such as cache misses and branch mispredicts, to assess the presence and severity of various performance issues. A fixed penalty (latency cycles) is assigned for each type of event. This methodology ignores the interaction between various events and the ability of modern microprocessors to hide latency using techniques such as out-of-order execution, pre-fetching and speculative execution. This results in micro-architectural performance events having varying penalty depending on the amount of latency that can be hidden, which in turn depends on the characteristics of the workload (e.g., instruction mix and present level of parallelism) and the interaction with other performance events (e.g., the presence or absence of other performance events).

This paper considers two important performance analysis questions:

- The "what" question: This question tries to identify the main performance issues or sources of potential performance improvements. This is important, as it can orient the effort of the performance analyst to optimize for specific performance issues (e.g., reduction of cache misses).

[*]The authors would like to thank the following people for their help with this work: Seth Abraham, Antonio C. Valles, Garrett T. Drysdale, James C. Abel, Agustin Gonzalez, David A. Levinthal, Stephen P. Smith, Henry Ou, Yong-Fong Lee, Alex A. Lopez-Estrada, Kingsum Chow, Thomas M. Johnson, Michael W. Chynoweth, Annie Foong, Vish Viswanathan

- The ”how much” question: This question tries to estimate the potential performance gain (e.g., percentage reduction in execution time) from mitigating a specific performance issue or a set of performance issues. This question is important as there may be several performance issues and one needs to decide which ones are most important and whether it is worth to trying to optimize for a specific issue.

In answering the previous two questions, it is important to take into account the potential interactions between different performance events. For example, if two events tend to occur at the same time, it is possible that the actual penalty incurred for one (e.g., Level 1 cache miss) depends on the occurrence or not of the other (e.g., DTLB miss). These interaction effects create non-linearities and can result in distinct performance models for different categories of workloads and even phases of a single workload [18].

We investigate the usefulness of machine learning regression techniques to construct accurate and useful performance models that can address the “what” and “how much” questions. In particular, we focus on models that can be used to diagnose potential performance issues and to estimate the potential gain from addressing one or more specific performance issues. Evaluation is performed on five different regression algorithms to compare the pros and cons of these algorithms and to determine which algorithm or class of algorithms is suitable for performance analysis.

The remainder of the paper is organized as follows. Section 2 discusses some of the related work. Section 3 presents the different regression algorithms used in this study. Section 4 describes in detail our experimental setup used for data collection. Section 5 presents our results. Section 6 concludes the paper.

2 Related Work

Recent years have seen several attempts to build models for performance analysis of processors. Unfortunately, most of these models fail to include many micro-architectural events and design space parameters, which leaves validity unknown when a large set of events and design parameters are present. This is mainly because these models require prior knowledge about significant events and parameters, and the required knowledge is gained from expensive, simulation-based sensitivity analysis. In addition to its prohibitive cost, simulation accuracy is questionable especially in the case of applications whose time varying behaviors are not easily represented in traces. Our work avoids these problems by relying on counts of a broad spectrum of processor events collected during the execution of the entire application, rather than those obtained through simulation. Such counts include, for example, misses in the various code, data and translation caches, branch mispredicts, load-store address overlaps, and many other events that can potentially reduce performance.

In [10], the authors propose a linear formula expressing the cycles-per-instruction (CPI) metric as a function of data and instruction cache misses, branch mispredicts and the ideal steady-state CPI. The performance penalty of cache misses and branch mispredicts is estimated using trace-driven simulation. The work in [20] extends [10] by including the effects of pre-fetching and resource contention in the model and uses a probabilistic approach to limit the required number of trace-driven simulation scenarios. These two approaches do not include other critical potential sources of CPI degradation such as DTLB and ITLB misses, various load blocks and the effects of unbalanced instruction mixes. More importantly, the two models do not account for the inherent interaction effects between various performance events, or for differing behaviors from application to application and often among different phases of the same application [18]. In contrast, this work establishes a classification of workloads or phases of workloads, and builds a model for each class, using measured performance data rather than simulation data.

In [7, 21], analytical models are used to study the effect of pipeline depth on performance for in-order and out-of-order processors. These two works use simulation-based sensitivity analysis to determine important model parameters. In [7], detailed superscalar simulation is used to determine the fractions of stall cycles for different pipeline stages and the degree of superscalar processing that remains viable. In [21], the authors use detailed simulations of a baseline scenario and scenarios with increased processor front-end width to determine the effects of micro-architecture loops (e.g., branch mispredict loops) on the performance. Again, these two models take into account only one aspect of the performance analysis. Our model, on the other hand, considers the processor performance as a whole while including many potential sources of performance degradation.

Several statistical techniques have been used to limit the required number of simulation runs for design space exploration needed during the design phase of new processors. In [5, 4], principal component analysis is used to limit design space exploration by identifying key design space parameters and computing their correlations. Placket and Burman fractional design is used in [24] to establish parameter prioritization for sensitivity analysis. The authors model high and low values of a set of N design parameters using only $2N$ simulations focusing on parameters with high priority.

In [6], the authors define interaction cost to account for the interaction between two different micro-architectural performance events. The authors design new hardware to enable sampling workload execution in sufficient detail to construct representative dependency graphs to be used for the computation of the interaction cost. Our approach also takes into account the interaction between various micro-architectural events. However, we propose the handling of the interaction cost in a statistical manner without the requirement of dedicated new hardware.

3 Methodology: Regression Algorithms

This section describes the different regression approaches used in this paper. Regression consists of fitting a model that relates a dependent variable Y to a set of independent predictors X_1, X_2, ..., X_k. The functional form of the model can be estimated using training samples from the unknown underlying distribution. In this study, we compare the merits of five different regression algorithms in the context of performance analysis: (1) Multi-linear regression [17], (2) Artificial neural networks [13], (3) Locally weighted linear regression [2], (4) Model trees [22] and (5) Support vector machines [15, 19]. These algorithms are described briefly below.

3.1 Multi-Linear Regression

Linear regression [17] is based on the assumption of a linear relationship between the dependent variable Y and its predictors X_1, X_2, ..., X_n. Linear regression offers simple and easily interpretable models. However, it can result in inaccurate models that predict poorly in the presence of a nonlinear or non-additive relationship. Due to the complexity of micro-architectural event interaction and varying event performance penalties, however, it is common for a nonlinear relationship to exist. In the linear case, the functional relationship between Y and its predictors is estimated by minimizing the residual sum of squares (RSS). For more details on multi-linear regression, the reader is referred to [17] or any classical statistics text.

3.2 Artificial Neural Networks

Artificial Neural Network (ANN) is a powerful method for generalized nonlinear regression. This class of algorithms is patterned after cooperative processing of information that is found in the biological world's neurons and networks of neurons [13]. A multilayer neural network consists of a number of neurons organized into an input layer, an output layer and a number of hidden layers. Units in the input layer take as input the information to be processed (values of the predictors in our case), while the output layer produces the prediction result. The first hidden layer receives as input the results of the units in the input layer and gives its results as inputs to the units in the next layer.

The training of an ANN establishes the input/output mapping in the form of connections between various units in the network. The training also computes the weights of input connections. The fitted model can be used to predict the values of the dependent variable Y for unseen data points. The ANN approach has two key benefits: (1) it has high prediction accuracy and (2) it does not require any prior knowledge of the form of the functional relationship between the dependent variable and the independent variables. It has the drawback, however, that the black-box nature of ANN thwarts interpretation of results and therefore prevents insight into the sources of performance degradation and the exact performance impact of the different micro-architectural events. In addition, the approach is known to be very sensitive to outliers.

3.3 Locally Weighted Linear Regression

Locally Weighted linear Regression (LWR) [2] is a "lazy" or instance-based learning technique. A new regression equation is fitted every time the model needs to predict on a new instance. This is in contrast with the other methods seen in this section where one regression model is built during the training phase and used with all test instances.

LWR combines linear regression and instance-based learning. Unlike regular linear regression, where one regression is performed on the full unweighted training set, LWR performs a new regression for each instance, weighting training instances based on their distance (e.g., Euclidean distance) from the specific test instance. The main advantage of LWR is its high flexibility, which makes it suitable for the approximation of nonlinear functions. The main disadvantage of this method, like all instance-based learning methods, is that it does not provide much insight into the global structure of the training data. This limits the interpretability of its output.

3.4 Model Trees: M5’

Model trees are a sub-class of regression trees [3], having linear models at the leaf node. In comparison with classical regression trees, model trees deliver better compactness and prediction accuracy. These advantages issue from the ability of model trees to leverage potential linearity at leaf nodes.

The model tree algorithm used in this work is based on M5’ [22], an optimized, open-source implementation of the classical M5 [16] algorithm. The input space is recursively partitioned until the data at the leaf nodes constitute relatively homogeneous subsets such that a linear model can explain the remaining variability. This divide-and-conquer approach partitions the training data and provides rules for reaching the models at the leaf nodes. The linear models are then used to quantify, in a statistically rigorous way, the contribution of each attribute (e.g., micro-architectural events here) to the overall predicted value (e.g., performance in this case). A powerful aspect of the prediction model arrived at in this way is that it is interpretable, in contrast with other machine learning approaches, such as neural networks.

3.5 Support Vector Machines

Support Vector Machines (SVM) [15] are a combination of instance-based and numeric modeling learning. The idea behind support vector machines is finding instances, called support vectors, that are at the boundary of the classes and creating linear functions that discriminate them as widely as possible. The biggest advantage of vector machines is that they can use linear, quadratic or higher order models to represent nonlinear boundaries between classes. This is in contrast to basic linear models that only represent linear boundaries. To construct nonlinear boundaries with linear models, support vector machines use nonlinear mapping, where the instance space is transformed, allowing a linear model to represent a nonlinear model in the previous space.

The Sequential Minimal Optimization algorithm (SMO) has been shown to be an effective method for training SVM on classification tasks defined on sparse data sets. SMO differs from most SVM algorithms in that it does not require a quadratic programming solver. The technique used here is a generalization of SMO by Shevade et al [19] to handle regression problems.

The main benefit of using SVMs is that they are robust against overfitting. Like ANNs, a problem with applying this technique to analysis of processor performance is that its black-box nature prevents insight into sources of inefficiencies. In addition, training SVMs is particularly slow. It took the authors more than 10 hours to train a model with performance data. In contrast, training model trees (M5’) on the same dataset using the same hardware required less than 10 minutes.

4 Experimental Setup

In this section, we describe the experimental setup we used to collect the necessary training data.

4.1 Platform

The data used in this study is collected on an Intel® Core™2 Duo processor-based desktop platform. The test machine has a speed of 2.4 GHz and 1GB of memory. The memory subsystem consists of a two-level cache. Each core has a 32 KB, level- one instruction cache and a separate level-one data cache of the same size. The two cores share a unified level-two cache of 4 MB. For more details on Core™2 Duo processor architecture, the reader is referred to [9, 8]. The data collection platform is running a Microsoft® Windows™XP 64 bit operating system.

4.2 Data Collection Methodology and Tool

The data was collected using an internally-developed tool. This tool is similar to the Intel VTune Performance Analyzer, but it collects data in counting mode. The counting mode obtains the values of a set of micro-architectural events of interest every time a threshold count is reached for another reference event. In particular, we divide the execution sequence into sections of equal numbers of instructions retired and collect the counts of various micro-architectural events for each section, using instructions-retired as the reference event.

Dividing the execution sequence into fine-grained sections in the above manner is done to capture the phase behavior of the workload [18]. In general, we expect that several phases, each with distinct performance characteristics, are present in the workload. Dividing the execution sequence into multiple sections increases the probability of capturing these phases.

4.3 Micro-Architectural Events

The Core$^{\text{TM}}$2 Duo architecture implements processor counters for multiplexed collection of information about several hundred micro-architectural events, that cover different aspects of the processor's behavior. Of these, a significant fraction of events can be excluded from consideration simply because they do not have a performance impact or do not arise except due to error conditions. Of what remains, it is still impractical to collect counts on a majority of events due to the small number of multiplexed counters. To offset these practical difficulties, it was necessary to pre-select a subset of events so a subset of 21 events was identified as candidates likely to be most relevant in the performance analysis. This apparently ad-hoc choice was purely pragmatic and revisable; happily, the prediction accuracy of the model, shown in Section 5, suggests it was on target. The chosen set of events represents the execution time and various performance-related micro-architectural events characterizing the instruction mix, the memory sub-system, the branch prediction accuracy, the data and instruction translation lookaside buffers and other known potential sources of performance degradation.

4.3.1 Execution Time

The execution time is the number of unhalted CPU clock cycles that the workload takes to execute, measured by the event CPU_CLK_UNHALTED.CORE, and considered the primary performance metric in this study. Workload sections consisting of equal numbers of instructions retired have radically different execution times. This event is used to derive the CPI (cycles per instruction) which constitutes our dependent variable.

4.3.2 Instruction Mix

While each section comprises a fixed number of instructions retired, the instruction mix can change from section to section. Different instruction mixes can give rise to different performance issues (e.g., data cache misses can only be caused by memory referencing instructions). In addition, different types of instructions execute on different functional units and stress different resources. For instance, Core$^{\text{TM}}$2 Duo architecture can retire up to four instructions per cycle, but these four instructions cannot contain more than one store instruction. This means that a high percentage of store instructions results automatically in a lower average number of instructions retired per cycle (longer execution time) even if there is no other performance issue. For the analysis, the retired instructions are divided into four different groups:

- Load instructions: the number of load instructions, counted using the INST_RETIRED.LOADS event.
- Store instructions: the number of store instructions, counted using the INST_RETIRED.STORES event.
- Branch instructions: the number of branching instructions counted using the BR_INST_RETIRED.ANY event. In this study, it is further divided into correctly predicted and mispredicted branches as will be discussed shortly.
- Other instructions: all other instructions, counted by subtracting the above three counts from the total number of instructions retired. In particular, this category includes both integer and floating point instructions.

4.3.3 Branch Related Events

The distribution of branches between predicted and mispredicted is critical, as each mispredicted branch forces the execution pipeline to be flushed and the fetch engine to be restarted at the correct branch target and costs up to a few dozen cycles.

- The number of mispredicted branch instructions is counted using the event BR_INST_RETIRED.MISPRED
- Subtracting the above from the total, i.e., (BR_INST_RETIRED.ANY - BR_INST_RETIRED.MISPRED) yields the number of correctly predicted branches.

4.3.4 Memory Subsystem Events

Load or Store instructions that miss in caches tend to have a profound impact on performance. We collect data on the number of misses occurring at various caches within the memory subsystem.

- The number of level 1 data cache misses is counted using the event MEM_LOAD_RETIRED.L1D_LINE_MISS. This does not double-count a cache line that is missed while still being brought into L1 cache as a result of a previous cache miss.
- The number of level 1 instruction cache misses. This count is obtained using the event L1I_MISSES.
- The number of level 2 cache misses is counted using the event MEM_LOAD_RETIRED.L2_LINE_MISS. In Core™2 Duo, the L2 cache is shared between the two cores and so this event counts both data and instruction misses in the level two cache.

4.3.5 Translation Lookaside Buffers Events

Data and instruction translation lookaside buffers (DTLB and ITLB) are critical resources for efficient execution across nearly all workloads. Several events are used to monitor the DTLB and ITLB stresses that arise during a specific workload or sections of it.

- The number of load accesses that miss the first level DTLB (L0 DTLB) is counted using the event DTLB_MISSES.L0_MISS_LD.
- The number of load accesses that miss the last level DTLB is counted using the event DTLB_MISSES.MISS_LD.
- The number of non-speculative load accesses that miss the DTLB, a subgroup of the previous event, is counted using the event MEM_LOAD_RETIRED.DTLB_MISS.
- The overall number of DTLB miss events which arise for any reason (i.e., due to loads, stores and hardware initiated memory references, including speculative operations) is counted using the event DTLB_MISSES.ANY.
- The overall number of retired instructions missing the ITLB is counted using the event ITLB.MISS_RETIRED.

4.3.6 Other Events

A number of other events often indicate potential performance issues.

- Load block related events: The Core™2 Duo processor uses memory disambiguation [9, 8] to maximize concurrency among loads and stores that don't intersect. In certain cases memory disambiguation fails, leading to different types of load blocks, depending on what causes the failure. LOAD_BLOCK.STA counts the number of load instructions blocked because of a preceding store instruction to an address that is not yet known. LOAD_BLOCK.STD measures the number of load instructions blocked because of a preceding store to the same address when the data to be stored is not yet known. LOAD_BLOCK.OVERLAP_STORE counts the number of load operations blocked because of an actual datum-width overlap with a preceding store, or because of an ambiguous overlap from page aliasing in which the load and a preceding store have the same offset but into different pages. Generally, these load block events can be avoided by increasing the distance between load and store instructions.
- Split events: Accesses that are not aligned to natural type-boundaries of data often cause additional cycles to complete, as the detection of potential conflicts with previous accesses may in general require blocking the current access until previous memory operations have retired. MISALIGN_MEM_REF counts the number of memory reads or writes that cross an eight-byte boundary. L1D_SPLIT.LOADS counts the number of load operations from the level 1 cache that span two cache lines. L1D_SPLIT.STORES counts the number of store operations to level 1 cache that span two cache lines.
- ILD_STALL: This event counts the number of instruction length decoder stall cycles due to a length changing prefix [9, 8]. Normally, instruction decoding takes one cycle; in the presence of a length changing prefix, it requires 6 cycles.

4.4 Data Pre-Processing

The Intel® Core^{TM}2 Duo architecture has five performance counters, which means that up to five micro-architectural events can be monitored simultaneously. However, three of these counters are fixed to always monitor the following events: "CPU_CLK_UNHALTED.CORE", "INST_RETIRED.ANY" and "CPU_CLK_UNHALTED.REF". As a result, there are only two reconfigurable performance counters, while our study requires data collection on about 20 different performance events. To work around this problem, it was decided to run each workload 11 times to collect the values of the required number of events for each workload section.

While this multiple-run approach is attractive as it allows seemingly simultaneous collection of data on all the necessary events, it has its own limitations. For instance, we observed a certain amount of variability from run to run. This can result from the presence of different operating system processes executing on the machine in addition to our workload. In our study, process affinitization was used to limit this variability. In addition, outliers with large variability were identified and removed from the data set. On the basis of several pilot tests, it was decided to use a 5% cutoff-threshold on variability; that is, workload sections for which the standard deviation for execution time from the 11 runs was higher than 5% of the mean were removed. Multi-run data collection is illustrated in Figure 1. Future work will involve the testing of event multiplexing [12] as an alternative.

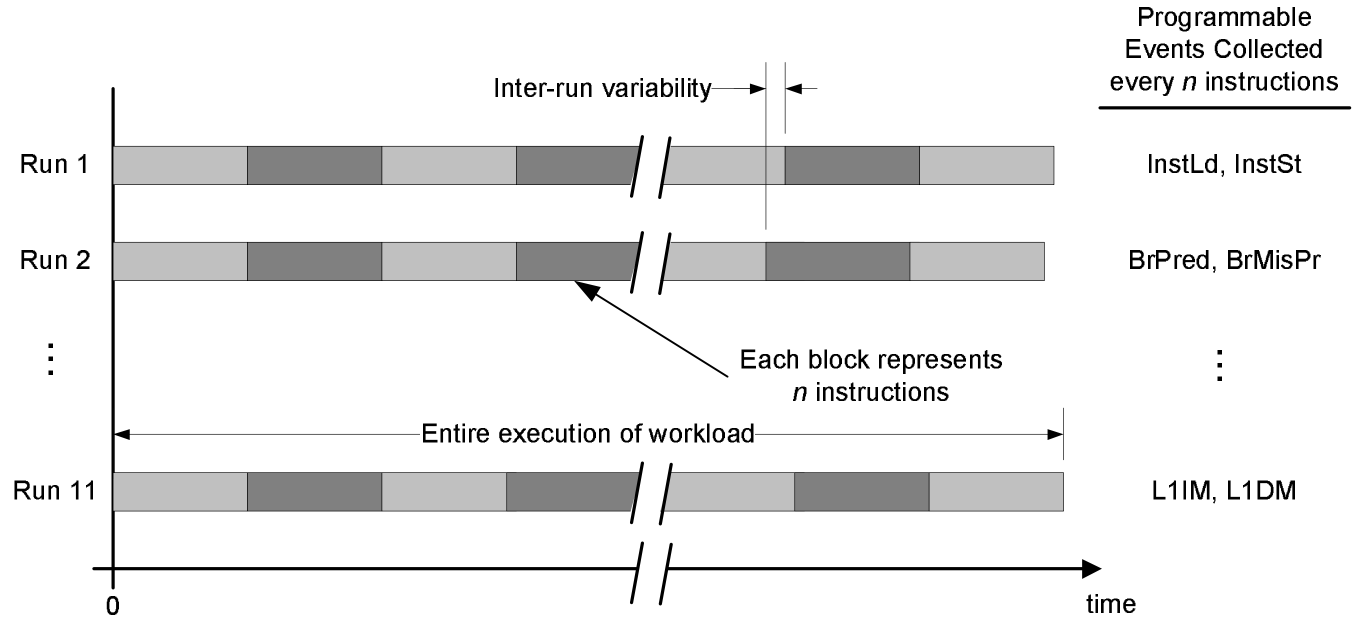

Programmable Events Collected every n instructions
Inter-run variability
Run 1
InstLd, InstSt
Run 2
BrPred, BrMisPr
Each block represents n instructions
Entire execution of workload
Run 11
L1IM, L1DM
0
time

Figure 1. Multi-run data collection approach

The data is normalized by the number of retired instructions. For example, instead of the execution time as the dependent variable, the CPI (cycles per instructions) is used. CPI is computed as the execution time (number of unhalted clockticks) divided by the number of retired instructions. Similarly, the number of level 2 cache misses per retired instruction and the number of branch mispredicts per retired instruction are used instead of the raw counts of the corresponding events.

Table 1 indicates the short names used for various variables (event counts per retired instruction) along with the corresponding events and a short description.

4.5 Workloads

The data is collected on a subset of SPEC CPU2006 workloads [1], as limiting the set of workloads was necessary to restrict the size of the data set due to limitations of the WEKA tool [23] used in this study. While the use of a limited set of workloads can affect the generalization of the proposed approach, it is argued that by dividing each workload into many sections we can capture more different performance behaviors. This increases the variance of our data, which provides higher

Table 1. Selected metrics used in this study

Metric	Corresponding event	Description
CPI	CPU_CLK_UNHALTED.CORE	CPU clock cycles per instruction
InstLd	INST_RETIRED.LOADS	Loads per instruction
InstSt	INST_RETIRED.STORES	Stores per instruction
BrMisPr	BR_INST_RETIRED.MISPRED	Mispredicted branches per instruction
BrPred	BR_INST_RETIRED.ANY – BR_INST_RETIRED.MISPRED	Correctly predicted branches per instruction
InstOther	INST_RETIRED.ANY – (INST_RETIRED.LOADS + INST_RETIRED.STORES + BR_INST_RETIRED.ANY)	Non-branch and non-memory instructions per instruction
L1DM	MEM_LOAD_RETIRED.L1D_LINE_MISS	L1 data misses per instruction
L1IM	L1I_MISSES	L1 instruction misses per instruction
L2M	MEM_LOAD_RETIRED.L2_LINE_MISS	L2 misses per instruction
DtlbL0LdM	DTLB_MISSES.L0_MISS_LD	Lowest level DTLB load misses per instruction
DtlbLdM	DTLB_MISSES.MISS_LD	Last level DTLB load misses per instruction
DtlbLdReM	MEM_LOAD_RETIRED.DTLB_MISS	Last level DTLB retired load misses per instruction
Dtlb	DTLB_MISSES.ANY	Last level DTLB misses (including loads) per instruction
ItlbM	ITLB.MISS_RETIRED	ITLB misses per instruction
LdBlSta	LOAD_BLOCK.STA	Load block store address events per instruction
LdBlStd	LOAD_BLOCK.STD	Load block store data events per instruction
LdBlOvSt	LOAD_BLOCK.OVERLAP_STORE	Load block overlap store per instruction
MisalRef	MISALIGN_MEM_REF	Misaligned memory references per instruction
L1DSpLd	L1D_SPLIT.LOADS	L1 data split loads per instruction
L1DSpSt	L1D_SPLIT.STORES	L1 data split stores per instruction
LCP	ILD_STALL	Length changing prefix stalls per instruction

representativeness of the data set. It must be mentioned here that nothing in the proposed approach restricts working only on this specific set of workloads. Future work includes data collection on commercial workloads to train the solution on a richer data set.

A total of several hundred thousand data points is obtained by dividing the workloads into distinct sections. Due to the large data set, a subset of the data is chosen randomly in a way that maintains the statistical distribution of the overall population. This is done using the "stratified remove fold" statistical technique implemented in WEKA. The resulting subset contains $27,448$ data points and maintains the same statistical distribution as the original application.

We used the following workloads with varying input files (reference and training input files delivered with SPEC).

- 401.bzip2: A compression benchmark based on Julian Seward's bzip2 version 1.0.3 with the exception that SPEC bzip2 does not perform any I/O operation except at the reading the input file.
- 403.gcc: A compiler benchmark based on gcc version 3.2. The benchmark modifies the original gcc compiler to perform more inlining and spend more time analyzing the source code input, which induces higher memory usage.
- 429.mfc: This benchmark is derived from a program to optimize single-depot vehicle scheduling in a public transportation system.
- 436.cactusADM: This benchmark is based on the Cactus computational framework used for solving the Einstein evolution equations in the ADM 3+1 formulation.
- 447.dealII: This benchmark is based on the C++ deal.II library used for adaptive finite elements and error estimation.
- 454.calculix: This benchmark is based on a finite element software used for linear and nonlinear three-dimensional structural applications.
- 458.sjeng: This benchmark is based on a program that plays chess and other chess variant games such as drop-chess.

5 Implementation and Results

For the purpose of this study, we use the open-source implementation available in the WEKA software package [23]. WEKA offers a unified framework for comparing the different algorithms described in Section 3. Each regression algorithm is trained on the dataset described in the previous section to obtain a performance model. To avoid any expert knowledge, we decided to use the default parameter for each algorithm. The only exception to this rule is the case of M5', where we report results for two different sets of runs. For the first set, we employ default algorithm parameters and call it the M5Default. To improve intepretability and achieve better compactness, for the second set, we forgo the smoothing technique that is otherwise applied by default and increase the minimum number of instances in leaf nodes and this in turn causes a small reduction in prediction accuracy for the second set. Results for the second set are labeled as M5Modified.

For performance analysis, two important qualities are desired in fitted models: model interpretability and performance predictability. Interpretability refers to the ability to explain the predicted value. For example, it is important to know why the predicted CPI of one class of work is 3, while that of another is 0.5. The understanding that interpretability creates allows one to diagnose performance issues (e.g., the low performance is due to cache misses) and to guage the possible gain from addressing specific performance issues. Prediction accuracy, measured in multiple ways, provides bounds on modeling errors and is important as a primary measure of confidence in the model. A physical systems exact performance may be readily measured and in such a case, the role of prediction in reaching a performance estimate may be marginal; but, a high prediction accuracy confirms that the dominant characteristics of the physical system are faithfully abstracted by the model.

5.1 Interpretability of the Algorithms

Among the five algorithms, only model trees and multi-linear regression give easily interpretable results. Equation 1 gives the multi-linear regression performance model:

$$\begin{aligned} CPI = 0.64 + 318.05 * LCP + 181.62 * ItlbM - 0.13 * DtlbL0LdM + \\ 11.40 * DtlbLdM - 6.14 * DtlbM + 1.74 * DtlbLdReM + 8.77 * L1DM - 4057.39 * L1DSpSt + \\ 198.25 * L2M + 6.67 * L1IM + 0.95 * LdBlSta + 1.62 * LdBlOvSt \\ -96.66 * LdBlStd + 2.86 * InstSt + 0.19 * InstLd + -832.07 * L1DSpLd + \\ 91.33 * MisalRef - 1.76 * BrPred + 18.41 * BrMisPr - 0.22 * InstOther \end{aligned} \tag{1}$$

This equation can be easily interpreted. It tells us, for example, that a level 2 cache miss (L2M) costs on average about 200 cycles and that a branch mispredicts costs about 18 cycles. However, the equation exposes several model anomalies. In particular, the negative coefficients in front of several micro-architectural events known to impede performance are counterintuitive. It is well known that store and load splits (L1DSpSt and L1DSpLd), DTLB misses (DtlbM) and load blocks (LdBlStd) degrade performance considerably if they occur, even in moderate frequency. The fact that the model tells otherwise can be the result of the inherent interactions between the different micro-architectural events. These interactions cannot be captured by a linear model. In addition, the approach of one model fits all is not realistic as previous research [18] proved that distinct performance behaviors or phases can exist even within a single workload. This translates to the necessity of using different performance models for different categories of workloads and phases within each workload.

Model trees seem to provide the best interpretable performance model. Each leaf node in a model tree represents a distinct workload or sections of workload class. The number in parentheses indicates the percent of the training set that falls into the corresponding leaf. The performance within each class is explained by a linear model. Interaction terms are captured in the tree structure, which can show for instance, that the performance effect of a given density of DTLB misses over an interval differs according to whether or not a significant number of L2 cache misses occur in the same interval. Figure 2 presents the performance analysis model tree obtained from applying M5' to the training set. This tree structure provides key insights to performance analysts. For example, at the root node of the tree, we can immediately see that the model identifies the level 2 cache misses (L2M) as the single event that most strongly affects performance. Among the events used in this study, L2 cache misses are known to be the longest latency event and so this result is highly intuitive. For more details on the interpretation of the model tree and its use for workload performance analysis, the reader is referred to [14].

The three other regression algorithms (LWR, ANN and SMOreg) are black-box techniques. Their outputs do not offer clear insights into the potential sources of performance degradation and, hence, the obtained models cannot be used to guide the performance optimization efforts as can be done with model trees.

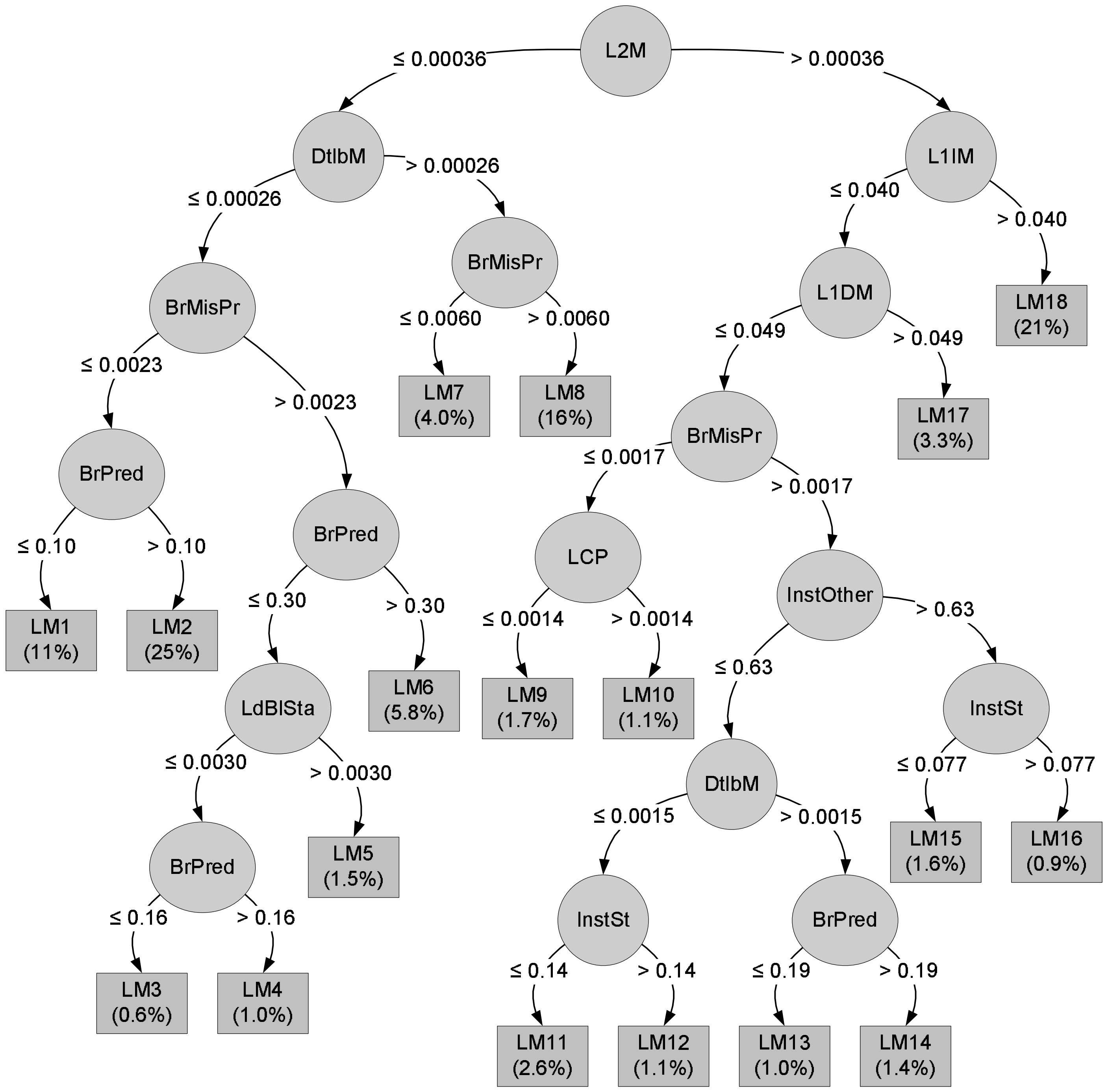


Figure 2. Performance analysis tree

5.2 Prediction Accuracy

To evaluate the prediction accuracy of the different algorithms for the performance data, we used 10-fold cross validation [11]. This technique consists of dividing the overall data in 10 disjoint subsets, or folds. Each algorithm is then trained using 9 of the subsets and evaluated using the tenth subset. The process is repeated 10 times and each time, a different subset is used for testing and the remaining 9 subsets are used to train the model. The algorithm is evaluated by averaging the prediction metrics from the 10 different models. Several prediction metrics can be employed and we use the following common metrics :

- The Correlation Coefficient: This metric is based on the standard correlation coefficient and measures the extent of linear relationship between predicted (P) and actual (A) values. It is a dimensionless index that ranges from -1 to 1 with 1 corresponding to ideal correlation. The correlation coefficient C is given by:

$$C = \frac{Cov(P, A)}{\sigma_p \sigma_a}. \tag{2}$$

  where Cov(A,P) is the covariance between the predicted and the actual values, while σ_p and σ_a are their respective standard deviations.

- Root Mean Squared Error (RMSE): This error measure is used in the determination of confidence intervals. Measured in the same unit as that of the predicted quantity (in this case CPI), it ranges from 0 to infinity with 0 corresponding to the ideal situation. It is computed as:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (p_i - a_i)^2}{N}} \tag{3}$$

  where p_i and a_i are the predicted and actual CPIs for i^th test instance and N the number of instances.

- Mean Absolute Error (MAE): This error measure is similar to RMSE, except that it uses absolute error values instead of the squared errors, i.e.,

$$MAE = \frac{\sum_{i=1}^{N} |p_i - a_i|}{N}. \tag{4}$$

- Root Relative Squared Error (RRSE): The relative squared error is relative to what it would have been if a naive predictor had been used. In particular, this simple predictor is just the mean of the actual values. It takes the total squared error and normalizes it by dividing by the total squared error of the simple predictor. It is given by:

$$RRSE = \sqrt{\frac{\sum_{i=1}^{N} (p_i - a_i)^2}{\sum_{i=1}^{N} (\hat{a} - a_i)^2}} \tag{5}$$

  where $\hat{a}$ is the mean of the actual CPI values.

- Relative Absolute Error (RAE): This error is computed in a similar way to RSE. It is given by:

$$RAE = \frac{\sum_{i=1}^{N} |p_i - a_i|}{\sum_{i=1}^{N} |\hat{a} - a_i|}. \tag{6}$$

  The value of RAE ranges from 0% to 100% with 0 being the ideal situation.

Table 2 gives the evaluation results for the different algorithms compared in this study.

The results indicate that all nonlinear regression methods have good prediction accuracy. All except the locally weighted linear regression result in a correlation coefficient exceeding 0.95 and a relative absolute error below 10%. It is also clear that the model tree methods, M5' with the default parameters and the modified version, demonstrate very competitive prediction accuracy. In the case of the modified algorithm, where prediction quality was traded off for more compact tree and improved interpretability, the predictions are still highly accurate.

Table 2. Prediction accuracy for different algorithms

Algorithm	Correlation	RMSE	MAE	RRSE	RAE
LinReg	0.9832	0.1745	0.0844	18.2284	11.342
ANN	0.9955	0.0902	0.0412	9.4254	5.5365
LWR	0.9119	0.3927	0.1931	41.0354	25.9568
M5'Default	0.9962	0.0836	0.0245	8.7301	3.2881
M5'Modified	0.9845	0.1676	0.0582	17.5153	7.8302
SMOreg	0.9758	0.2143	0.0702	22.3865	9.4344

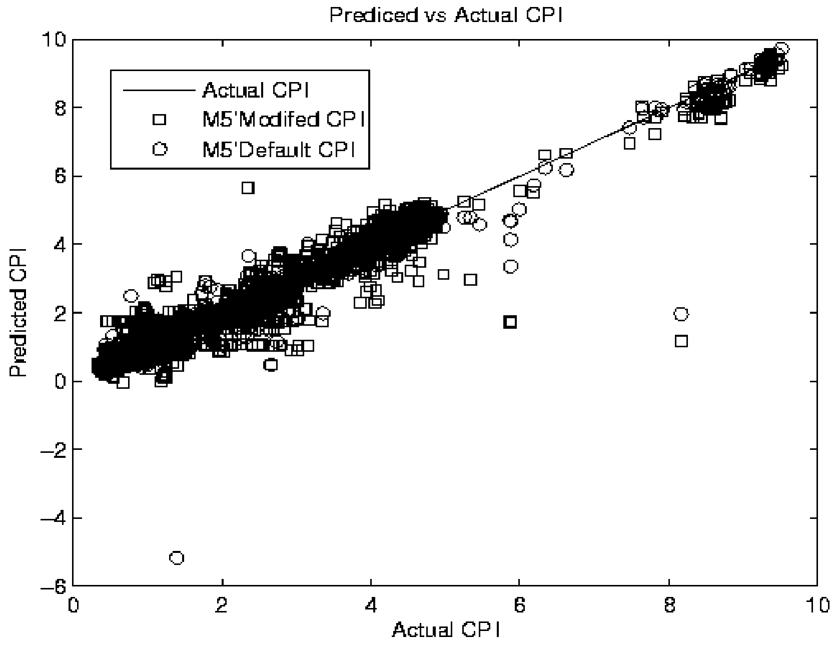


Figure 3. Predicted CPI vs. Actual CPI Using Default and Modified M5'

To illustrate the prediction accuracy of the different approaches, Figures 4 and 5 plot the predicted CPI versus the actual CPI for the cross-validation data. Note that the prediction is performed on data points in the test fold. In other words, the prediction on each data point is performed using a model that was built on training data that does not include the data point. Figure 3 plots the CPI predictions using the default and modified implementations of the M5' algorithm. Clearly, the two algorithms produces CPI predictions that are very close to the actual CPI values. Note, however, that the default implementation of M5' seems to perform poorly when it comes to outlier cases. The figure shows a case where the predicted CPI is negative.

Figure 4 plots the predicted against the actual CPI for the ANN and SMOreg (extension of SVM) algorithms. As noted earlier, artificial neural networks shows exceptionally good prediction accuracy. In the case of SVM-based regression, the predicted CPI shows a negative bias for large CPI. This seems also to be the case for LWR (locally weighted linear regression) and multi-linear regression as shown in Figure 5.

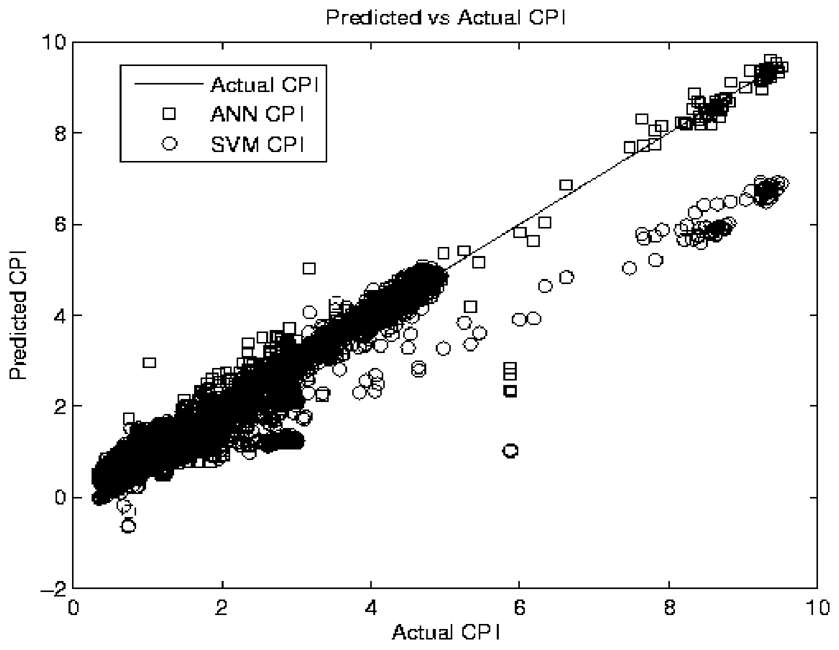


Figure 4. Predicted CPI vs. Actual CPI Using ANN and SVM

It is clear from these figures that the modified implementation of M5' does not sacrifice prediction accuracy for interpretability. It shows very competitive overall prediction and resilience to outlier cases.

6 Summary and Conclusions

In this paper, we presented several regression algorithms and assessed their appropriateness for computer architecture performance analysis for workload tuning. The results show that the M5' algorithm representing model trees demonstrates high prediction accuracy and excellent interpretability. These two properties allow the exploitation of the resulting models for the identification of main performance issues and the quantification of the potential gain from addressing each. The work presented here has many potential applications in the area of micro-architecture and software performance analysis. In particular, the model tree approach is worth investigating for use in processor design space exploration.

References

[1] Standard performance evaluation corporation. SPEC CPU benchmark suite. http://www.specbench.org/osg/cpu2006, 2006.
[2] C. G. Atkeson, A. Moore, and S. Schaal. Locally weighted learning. *Artificial Intelligence Review*, 11:11–73, 1997.

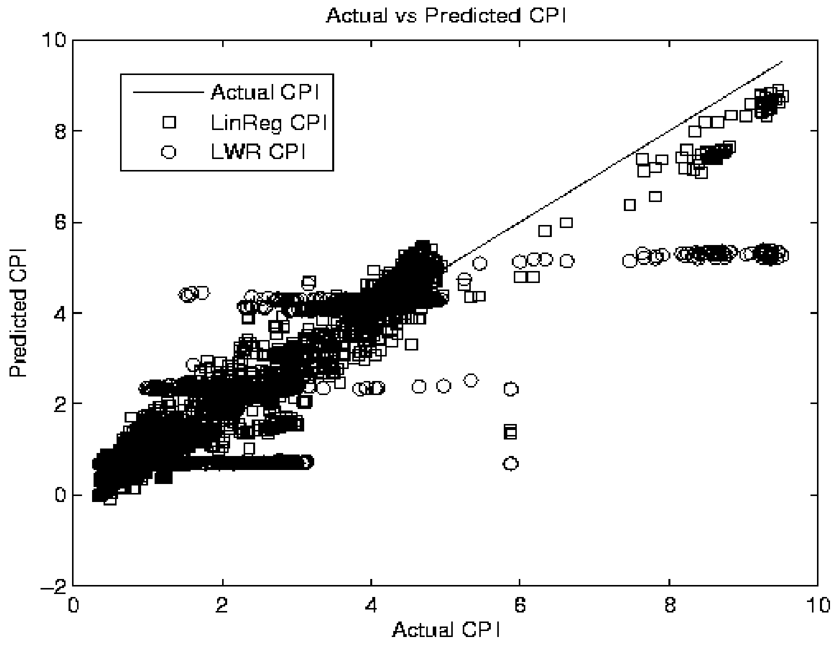


Figure 5. Predicted CPI vs. Actual CPI Using Multi-Linear Regression and Locally Weighted Linear Regression

[3] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth International Group, 1984.
[4] G. Cai, K. Chow, T. Nakanishi, J. Hall, and M. Barany. Multivariate power/performance analysis for high performance mobile microprocessor design. In *Proceedings of Power Driven Microarchitecture Workshop*, 1998.
[5] K. Chow and J. Ding. Multivariate analysis of Pentium Pro processor. In *Proceedings of Intel Software Developers Conference*, 1997.
[6] B. Fields, R. Bodick, M. Hill, and C. Newburn. Interaction cost and shotgun profiling. *ACM Transactions on Architecture and Code Optimization*, 1(3):272–304, 2004.
[7] A. Hartstein and T. Puzak. The optimum pipeline depth for a microprocessor. In *Proceedings of the International Symposium on Computer Architecture (ISCA'02)*, 2002.
[8] Intel. IA-32 intel architecture optimization: reference manual. http://www.intel.com/design/Pentium4/manuals/248966.htm.
[9] Intel. Intel 64 and IA-32 architectures optimization reference manual, to appear. http://developer.intel.com/design/-Pentium4/manuals/index_new.htm, 2006.
[10] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the International Symposium on Computer Architecture (ISCA'04)*, 2004.
[11] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of 14th International Joint Conference on Artificial Intelligence*, 1995.
[12] W. Mathur and J. Cook. Improved estimation for software multiplexing of performance counters. In *Proceedings of 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS05)*, 2005.
[13] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.
[14] E. Ould-Ahmed-Vall, J. Woodlee, C. Yount, K. A. Doshi, and S. Abraham. Using model trees for computer architecture performance analysis of software applications. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'07), to appear*, 2007.
[15] J. Platt. Using sparseness and analytic qp to speed training of support vector machines. In *Proceedings of Advances in Neural Information Processing Systems (NIPS'99)*, 1999.
[16] R. Quinlan. Learning with continuous classes. In *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence (AI'92)*, 1992.
[17] P. Rousseeuw and A. M. Leroy. *Robust Regression and Outlier Detection*. Wiley, 1995.
[18] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of 30th Annual International Symposium on Computer Architecture (ISCA'03)*, 2003.

[19] S. Shevade, S. Keerthi, C. Bhattacharyya, and K. Murthy. Hp caliper: A framework for performance analysis tools. *IEEE Concurrency*, 8(4), 2000.

[20] L. Simonson and L. He. Micro-architecture performance estimation by formula. In *Proceedings of the 5th International Workshop on Embedded Computer Systems: Architectures, MOdeling, and Simulation (SAMOS'05)*, 2005.

[21] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the International Symposium on Computer Architecture (ISCA'02)*, 2002.

[22] Y. Wang and I. Witten. Inducing model trees for continuous classes. In *Proceedings of the 9th European Conf. on Machine Learning, Poster Papers*, 1997.

[23] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 2000.

[24] J. Yi, D. Lilja, and D. Hawkins. A statistically-rigorous approach for improving simulation methodology. In *Proceedings of 9th IEEE Symposium on High Performance Computer Architecture*, 2003.

Branch Prediction with Bayesian Networks

Jeremy Singer, Gavin Brown, and Ian Watson

University of Manchester, UK

Abstract. This paper studies the architectural problem of *branch prediction*. We analyse the popular technique of *two-level adaptive prediction*, relating it to the state-of-the-art Machine Learning technique of *Bayesian Networks* (BNs). We show that a two-level predictor is an approximation to the BN formalism. This link allows us to explore the wider family of BN predictors. We investigate how to adapt BN techniques to operate within realistic hardware constraints, using the same primitive components that are present in existing branch predictors. We systematically study how performance is affected by these simplification. We aim to use these ideas to reduce the storage overhead of BN predictors without losing significant prediction accuracy. The key motivating factor is that storage required in two-level predictors grows exponentially with branch history length, whereas BNs provide a framework to reduce this overhead.

1 Introduction

There is an increasing trend to apply machine learning (ML) techniques to diverse prediction problems in computer systems. Recent examples include architectural simulation [1] and operating system message passing [2]. A challenging case appears to be the online learning problem of *branch prediction* since it requires (1) high accuracy, (2) low latency and (3) low implementation complexity.

1. High accuracy is essential, since predictors deployed in existing commodity processors achieve accuracy rates of over 97% on integer benchmarks [3]. However even slight improvements on this score are welcome, since the branch mispredict penalty is increasing all the time, with longer pipelines and speculative execution schemes relying on almost-perfect branch prediction.
2. Low latency generally means that the predictor must be able to supply a result in less than than a single processor cycle [4]. With the seemingly relentless increase of clock speeds, this latency requirement becomes ever more demanding.
3. Low complexity is necessary to ensure that the predictor designs can be fabricated in realistic transistor budgets, using low storage overheads and simple inputs that are trivially available in existing processor layouts.

1.1 Motivation

This paper relates a popular ML technique, *Bayesian Networks* (BNs), to existing branch prediction technology, *two-level adaptive branch predictors* [5], particularly the *GAg* and *gshare* schemes. We investigate how to adapt BNs to operate within realistic hardware constraints, by using the same primitive components that are present in existing branch predictors. We systematically study how performance is affected by these simplifications. We aim to use these ideas to reduce the storage overhead of two-level predictors without losing significant prediction accuracy. The key motivating factor is that storage required in two-level predictors grows exponentially with history length, whereas BNs provide a framework to reduce this overhead. The general community consensus is that longer history allows more accuracy, only the exponential growth rate of two-level adaptive predictors prevents longer histories from being used.

1.2 Contributions

This paper has four main contributions.

1. It simplifies the design of BNs so that they are suitable for hardware implementation.
2. It studies how the parameters of the BN model affect its prediction accuracy and storage capacity, the age-old tradeoff in a new context.
3. It discusses the theoretical relation between the GAg predictor and the *fully connected Bayesian Network*.
4. It explores whether BNs are a suitable alternative to GAg or gshare for branch prediction tasks.

2 Background

Any project that combines research from two distinct areas has the task of communicating with two distinct audiences, often with very different research objectives. In this work we attempt to use consistent terminology throughout, so after clarification, some ML concepts may be given architectural labels and vice versa. Section 2.1 describes the practicalities of two-level prediction from an architectural perspective. Section 2.2 outlines the theory of BNs from a ML perspective. Section 2.3 relates these two approaches to prediction, highlighting potential research issues.

2.1 Two-Level Adaptive Branch Prediction

Conventional branch predictors in the systems architecture community are generally implemented as lookup tables. This paper focuses on predictors composed of a global table indexed by a global history register. Yeh and Patt describe these as *GAg* schemes [5].

Each table entry is a 2-bit bimodal counter [6]. When the branch is taken, the appropriate counter is incremented until it saturates at 11_2. When the branch is not taken, the appropriate counter is decremented until it saturates at 00_2. The more significant bit is used to determine whether the predicted outcome is taken or not taken. The bimodal counter provides hysteresis, or the ability to remember general previous behaviour despite transient variations, for instance after the last iteration of a loop.

The index into this table of counters is derived from the outcomes of the most recently executed branches. Hence such schemes are known as *finite context method* predictors since they use a finite context of recent global branch history outcomes to construct the table index. This finite context is recorded in the *global history register* which operates like a shift register, shifting in the most recent branch outcome as the least significant bit after each branch instruction execution. In the simplest case (GAg), this global history is used as the table index directly, which means that the table must have 2^n entries where n is the length of the global history register. In a more complicated case, the global history may be XOR'd with low-order bits from the branch instruction address. This scheme is known as *gshare* [7]. Its XOR-based hashing function spreads the branch predictions more evenly throughout the table. This spreading alleviates the aliasing problem, which can occur when different branch instructions with different history patterns (although with a common suffix) map onto the same table entry.

2.2 Bayesian Networks

Bayesian Networks are a type of *probabilistic model*—a mathematical formalism within the field of Machine Learning, capable of representing and manipulating arbitrary probability distributions over arbitrary random variables. These are now commonly accepted as a state-of-the-art learning technique, finding applications in numerous domains from medical informatics to traffic control.

Bayesian Networks (BNs) are represented as directed acyclic graphs, where each node represents a different random variable. A directed edge from node X to node Y indicates that X has a direct influence on Y. This influence is quantified by the conditional probability $P(Y|X)$, stored at node Y. Nodes in a network can be of two types: *evidence (or attribute)* nodes, and *query (or class)* nodes.

For our purposes, BNs have multiple evidence variables X_1, X_2, $\ldots$, X_n and a single class variable C. In terms of branch prediction, the evidence is the value of previous branch outcomes. X_n is the outcome of the most recently committed branch and X_1 is the outcome of the oldest branch on record. The class node C represents the predicted next branch outcome. We adopt the standard encoding for branch outcomes, that 0 denotes 'not taken' and 1 denotes 'taken'. Thus all variables in the network are *boolean*.

The task of any branch predictor is to predict the next outcome, which can be rephrased as predicting the chance of each possible outcome (taken/not taken), given the evidence of previous outcomes. In the language of probability theory, this is the *posterior probability*, $P(C|X_1,\ldots,X_n)$. One approach to this is to attempt to devise a function that will directly estimate this value. This is the approach taken by the majority of predictors to date.

Alternatively, Bayes' theorem can be used to rearrange the problem,

$$P(C|X_1,X_2,\ldots,X_n) \;=\; \frac{P(C)\,P(X_1,X_2,\ldots,X_n|C)}{P(X_1,X_2,\ldots,X_n)} \qquad (1)$$

In practice, for a fixed branch history vector $(X_1,X_2,\ldots,X_N)$ the right-hand denominator is fixed. So we disregard this constant scaling factor. Now the numerator can be rearranged according to the rules of conditional probabilities, giving us a decision rule

```
if $P(C=1)P(X_1,\ldots,X_n|C=1) > P(C=0)P(X_1,\ldots,X_n|C=0)$
then predict taken
else predict not taken
```

The power of this approach is that we can make assumptions on the dependencies between the branches contained in the branch history buffer.

The simplest type of BN is called *Naive Bayes* (NB). This assumes that all the attributes are independent of each other. Figure 1 shows a NB network.

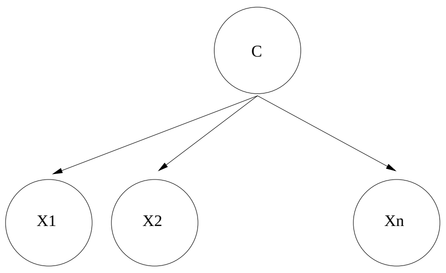


Fig. 1. Naive Bayes classifier

The independence property removes the dependence between each X_i and X_j, simplifying to:

$$P(C|X_1,X_2,\ldots,X_N)\;\alpha\;P(C)\prod_i P(X_i|C) \qquad (2)$$

Note that we employ upper case letters (X) to denote random variables, and lower case to denote specific values (x).

For a given history vector $x_1,x_2,\ldots,x_n$, we calculate $P(C=c|X_1=x_1,X_2=x_2,\ldots,X_N=x_n)$ using Equation 2 above, for each possible value of c (in our case either 0 or 1). We select as our prediction the more likely value, i.e. the one with the higher conditional probability. (Note that the scaling constant of proportionality is the same for all values of C.)

Although the NB classifier makes simplifying assumptions, it performs robustly for many prediction tasks.

Now we consider the space requirements of the NB classifier in terms of *number of probability values* that must be stored. Note that probabilities can be represented in different ways in hardware, as Section 3 explores. So, since there are two possible values for C, we require one probability value $P(C = 1)$. We can calculate $P(C = 0)$ as $1 - P(C = 1)$. Then, for each attribute X_i, each corresponding to a bit of global history, we require two conditional probabilities. $P(X = 1|C = 0)$ and $P(X = 1|C = 1)$. Again, we can derive $P(X = 0|C = c)$ as $1-P(X = 1|C = c)$. This means, in total, we must remember $2n+1$ probabilities, where n is the length of the global history register.

Friedman et al [8] show that the performance of a BN improves when *augmenting edges* are added between attributes. Recall that the NB classifier assumes all attributes are independent of one another. An *augmented naive Bayes classifier* relaxes this assumption by allowing edges between attributes.

A *tree-augmented naive Bayes classifier* (TAN) is a BN in which the class variable C has no parents and each attribute X_i has as parents the class variable and at most one other variable. Figure 2 shows a TAN network, in which each attribute X_i depends on the preceding attribute X_{i-1}. In terms of branch prediction, this means that a historical branch outcome depends on the prior branch outcome.

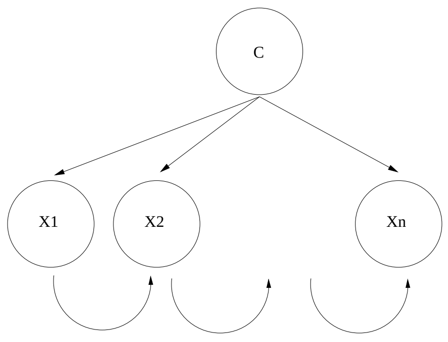


Fig. 2. Tree-augmented Naive Bayes classifier

The probability equation now looks like:

$$P(C|X_1, X_2, \ldots, X_n)\ \alpha\ P(C)\, P(X_1|C)\, P(X_2|X_1, C)\ \ldots\ P(X_n|X_{n-1}, C) \quad (3)$$

Whereas NB stores $2n + 1$ probability values for a history length n, TAN requires $4n - 1$ probability values, due to the additional dependences in the conditional probabilities. However the asymptotic space complexity is still $O(n)$. It is possible to add further augmenting edges to the BN, until eventually it becomes fully connected. In a *fully connected Bayesian network*, if there are n

attribute nodes and one classifier node, then each node is a member of n edge relations, either incoming or outgoing.

2.3 Relating Bayesian Networks and GAg

The GAg scheme effectively stores $P(C|X_1, X_2, \ldots, X_n)$ directly, using 2-bit bimodal counters as discretized estimates of conditional probability. This paper shows how we can use Bayes' rule to approximate this conditional probability, storing fewer probability values along the way. However, poor accuracy of predictors leads us to conclude that the NB network and its improvements do not capture sufficient dependence information for good prediction accuracy. This could be due to compound errors from approximations to probability multiplication.

3 Modifying Bayesian Networks for Hardware Implementation

The default NB model requires some adaptation to make it suitable for deployment in hardware. This section shows how a simplification of the NB model can be implemented using standard hardware components from existing branch prediction schemes, with a small storage overhead. Thus the simplified NB predictor satisfies two requirements from Section 1: *low latency* and *low implementation complexity.*

3.1 Representing the Probabilities

Equation 2 shows that a NB classifier requires the following probability values to be stored, or at least estimated: $P(C = 1)$ and each $P(X_i = 1|C = c)$ where i is bounded by the length of the global history register, and $c \in \{0, 1\}$. Recall that X_i is the ith bit of the global history register, storing the ith most recent branch outcome.

The accurate calculation of probability values requires frequency counts of events. For instance, to calculate $P(X_i = 1|C = 1)$ requires two event counters:

1. number of times that $X_i = 1$ and $C = 1$
2. number of times that $C = 1$

The second event counter will be reused for many probability calculations. However asymptotically the number of event counts scales as $O(n)$ with the history length n, in the same way as the number of probabilities.

Unfortunately, such event counts require an unbounded amount of storage. It is not possible to record probabilities as floating-point numbers since then it is not possible to know the relative significance of each new event, so the probabilities cannot be updated properly.

One workaround solution is to have a fixed window of recent events (like the global history register). It would be possible to store bounded counts in relation to this window size. This gives us a bound on storage capacity but there is another difficulty. The event counts have to be converted into probabilities using floating point arithmetic, which is almost certainly too complicated to include in the prediction unit. It may be possible to work in terms of logarithms, then the calculations become integer arithmetic but this is still too complex.

Discrete Approximations to the Probabilities A simpler scheme uses *2-bit bimodal counters*, inspired by the conventional table-based prediction schemes such as GAg and gshare. This effectively discretizes probability estimates. So $P(X_i = x|C = c)$ can be a value from the set $\{00_2, 01_2, 10_2, 11_2\}$. We use Smith's standard update scheme with increment/decrement and counter saturation at 00_2 and 11_2 [6]. For each bit of history, there are two counters $p_{i,0}$ and $p_{i,1}$ one for the case when c=0, the other for c=1. Counter $p_{i,0}$ will be updated when the branch outcome is 0. It will be incremented if $x_i = 1$, and decremented if $x_i = 0$. On the other hand, counter $p_{i,1}$ will be updated when the branch outcome is 1. Again, it will be incremented if $x_i = 1$, and decremented if $x_i = 0$. There is also a counter p_c, which is incremented every time the branch outcome is taken and decremented every time it is not taken.

So counter $p_{i,c}$ corresponds to the original probability value $P(X_i = 1|C = c)$. Just as we can compute $P(X_i = 0|C = c)$ using $1 - P(X_i = 1|C = c)$, we can compute the corresponding 2-bit bimodal counter as $11_2 - p_{i,c}$. For the rest of this section, we define function q as follows:

$$q_{i,c} = \begin{cases} p_{i,c} & \text{when } x_i = 1 \\ 11_2 - p_{i,c} & \text{when } x_i = 0 \end{cases}$$

This scheme provides a discretized estimate of posterior probabilities.

$q_{i,c}$	$P(X_i = x_i\|C = c)$
00_2	0
01_2	1/3
10_2	2/3
11_2	1

It might be possible to use these discretized probabilities to perform the actual probability calculation from Equation 2, using either lookup tables or simple binary algebra. However, the high frequency of 00_2 values ensures that most calculations generally result in 00_2 answers. The more satisfactory alternative to predict the outcome is to determine the more likely value (0 or 1) in the most significant bit of each bimodal counter indicated by Equation 2. So, for a history vector $x_1, x_2, \ldots, x_n$, we determine which of the following sets of counters has more top bits set: either (a) $\{p_c\} \cup \bigcup_i q_{i,1}$, or (b) $\{(11_2 - p_c)\} \cup \bigcup_i q_{i,0}$. If (a) has more top bits set than (b) then the predicted outcome is 1, otherwise the predicted outcome is 0.

The storage overhead of this 2-bit bimodal NB predictor is: two 2-bit counters for each bit in the global history register, plus one counter for C. This is a small storage overhead indeed. Moreover it scales linearly with history length, whereas the gshare storage scales exponentially.

Figure 3 shows how the simplified NB predictor is conceptually implemented in hardware. This schematic diagram shows that the set of 2-bit counters is arranged as a 3-d array, based on (c, i, x_i) where c is branch outcome, i is history bit index and x_i is the actual value of the ith history bit. In fact, since the limits of each dimensional index are fixed in advance, the 3-d array can be flattened to a linear vector. Also note that in the x_i dimension, we only need to store the 2-bit value v for when $x_i = 1$, since we can calculate the value for when $x_i = 0$ as $11_2 - v$. The same applies for the unconditional probabilities $P(C = 1)$ and $P(C = 0)$. Thus it is clear to see that for n bits of global history, the simplified NB predictor stores $2n + 1$ counter values.

The figure shows how to calculate the likelihood that the next branch outcome will be 0. We use counters from the $c = 0$ row and select between the $x_i = 0$ and $x_i = 1$ counters for each i based on the corresponding values in the global history register. We check the top bits for each selected counter and sum to see how many of these top bits are set. Then we do the same for the $c = 1$ row. We use a comparator (not shown in figure) to determine which outcome is more likely, and use this outcome as our prediction.

Predictor state update works in a similar way. We use the actual branch outcome to determine whether to update the $c = 0$ or $c = 1$ row. We update the $x_i = 1$ counters based on the values in the global history register—increment the counter if $x_i = 1$ or decrement if $x_i = 0$.

4 Evaluation Framework

We use the recently released second championship branch prediction framework (CBP2) [9] to evaluate our branch prediction implementations. Each implementation is coded in high-level C++, although in such a way that could easily be encoded in hardware. Thus arrays will map into indexed table lookups, integers will become bit strings, shifts and masks are used for bit selection and concatenation, and so on.

The default predictor is a simple gshare implementation. We compare all our predictors in this paper with this default gshare predictor. The framework includes a selection of real-world execution traces containing branch information. For each branch, the predictor is supplied with the branch instruction address and the branch target address. From these inputs and its internal state, the predictor must predict the branch outcome. Some short time later, the result of this branch is fed back to the predictor to enable it to update its state. The framework keeps track of the prediction accuracy and reports this at the end of the trace. Traces cover programs from benchmark suites including SPEC INT 2000 and SPEC JVM 98. The final result for a trace is reported in units of *MPKI*, which is mispredicts per 1000 instructions. The final result for all the programs is taken as the arithmetic mean of all individual traces. We modify the framework to report results in terms of percentage of mispredicted branches,

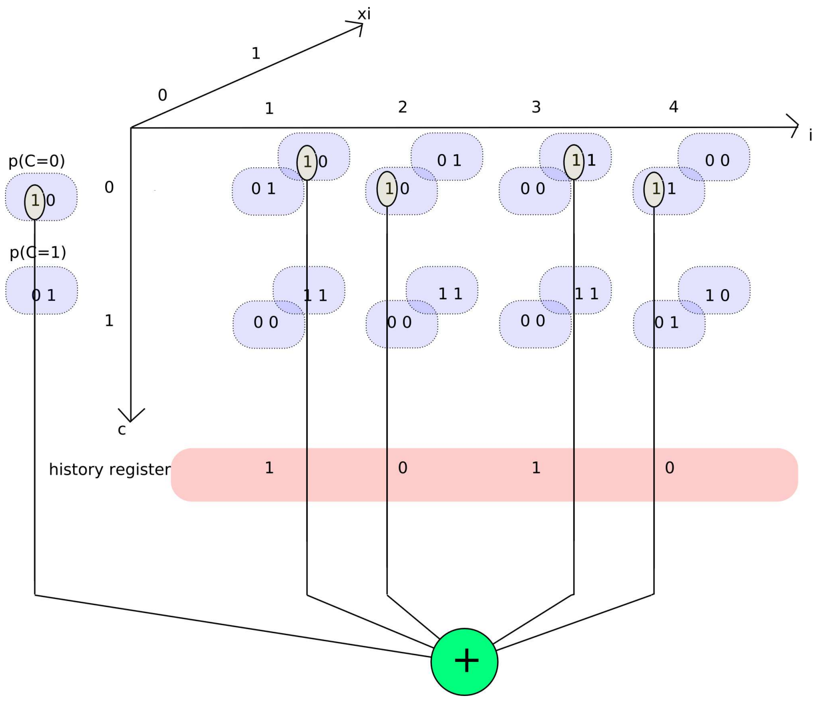


Fig. 3. Schematic diagram of simplified Naive Bayes predictor, calculating the likelihood that the next branch outcome will be 0 given the history 1010. The complete calculation requires checking the likelihood that the next branch will be 1, and then selecting the larger probability as the predicted outcome.

which is the conventional metric for branch prediction accuracy. CBP2 is an industrial quality simulation framework for branch prediction. Its predecessor, CBP1, has been comprehensively analysed by Loh [10].

The CBP2 default gshare predictor achieves a score of 6.3 MPKI (just under 5% misprediction rate). In CBP1, the winning predictor achieved a score of 2.5 MPKI. Any new predictor will need to have comparable performance and hardware complexity if it is to be accepted as a realistic alternative to current implementations.

5 Predictor Comparison: unbounded versus bounded NB

This section investigates how the NB simplification affects prediction accuracy. We implement two predictors, u-NB and n-NB. The u-NB model is an *unbounded* NB predictor, as described in Section 3.1. It keeps unbounded integer counts of all necessary events. It uses double-precision floating-point arithmetic to calculate all probabilities. It computes Equation 2 exactly to determine the most likely outcome. The n-NB model is the bimodal simplification as outlined in Section 3.1. It maintains probabilities as n-bit bimodal counters. These are updated with saturating increments or decrements. The prediction is made by examining top bits and choosing the outcome that has more top bits set.

Figure 4 shows the accuracy scores for u-NB, 2-NB and 4-NB on the CBP2 dataset, with different lengths of global history register. It is clear to see that accuracy improves as history length increases. This trend is apparent for both all three predictors. Although 2-NB tracks the performance of u-NB for short history lengths, the divergence increases with history length. The 4-NB predictor is more accurate than u-NB for all history lengths, and the performance improvement is sustained over longer history lengths than 2-NB. This is a most satisfactory result—our discrete approximation to the NB algorithm performs better than the original algorithm, for this dataset. This is because 4-bit counters provide enough *hysteresis* to avoid being upset by transient behaviour, but they are sensitive enough to forget the distant past history, in a way that u-NB cannot. Section 7.2 discusses this tradeoff further.

6 Predictor Comparison: NB versus gshare

This section compares the performance of our n-NB predictor with the standard gshare predictor. A fair comparison must ensure that predictors use equal storage capacities. We assume that the rest of the prediction logic to be roughly equivalent, so when we set the storage capacities to be equal then the predictors have equivalent implementation complexity.

For the 4-NB predictor from the previous section, with a history register length of h_b, there will be $2h_b + 1$ 4-bit bimodal counters, making a total of $8h_b + 4$ bits. In contrast, a gshare predictor for history length h_g will have a table of 2^{h_g} entries, each of which is a 2-bit bimodal counter, making a total of

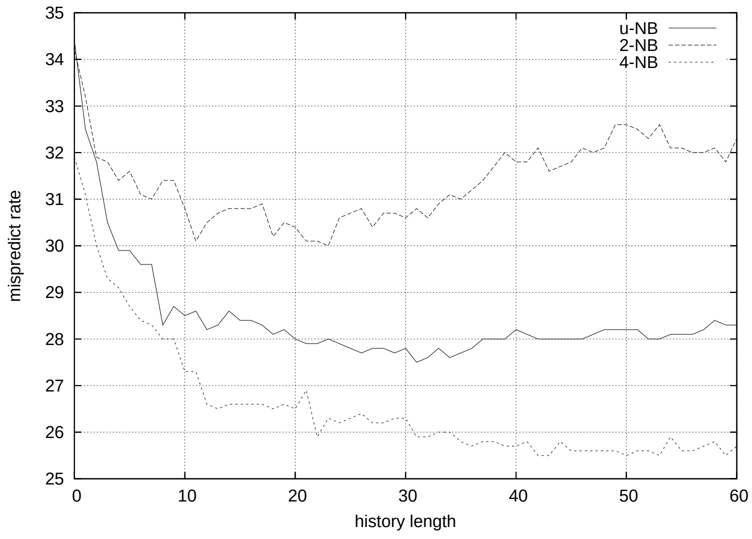


Fig. 4. Comparison of Naive Bayesian Predictors

2^{h_g+1} bits. A rearrangement of these equations shows that to have equal storage, the gshare predictor must have history length $h_g = \lceil 1 + log_2(2h_b + 1) \rceil$.

We investigate how these equations effect the predictors by comparing 4-NB predictors of varying history lengths with equivalently sized gshare predictors (with correspondingly shorter history lengths according to the above equation). Figure 5 shows the difference in performance between equally sized 4-NB and gshare predictors. Note that gshare is the clear winner and the performance difference grows with history length.

Thus it is clear that the simple adaptation of NB is inferior to gshare, not a suitable candidate for deployment in real processors. The next section investigates how to close this gap between n-NB and gshare, by considering different improvements to the n-NB prediction scheme. We hope to retain the branch predictor characteristics (from Section 1) of low implementation complexity and low latency, while achieving *high accuracy.*

7 Exploring the Bayesian Predictor Family

There are various parameters in the n-NB predictor model that may be tuned in order to improve the accuracy of predictions. This section considers tuning the global history register length (Section 7.1), the bimodal counter length (Section 7.2), the set associativity (Section 7.3) and the connectedness of the BN (Section 7.4).

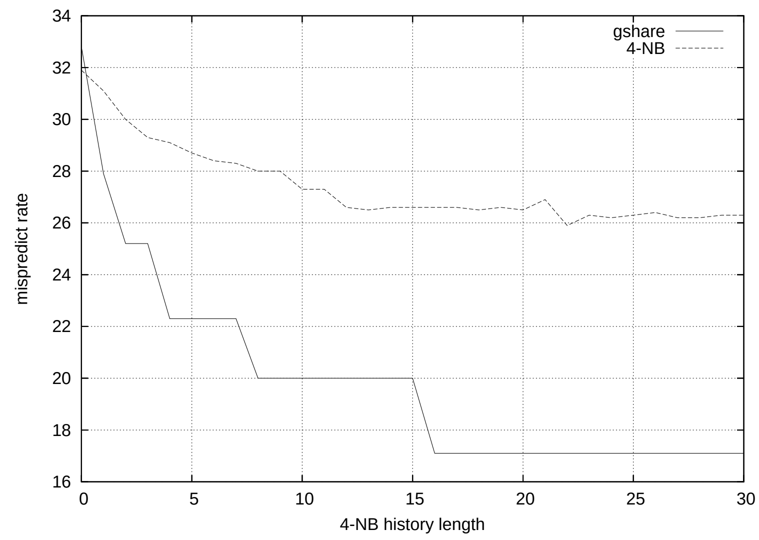


Fig. 5. Comparing performance of equally sized NB and gshare predictors

7.1 History Length

The standard method to increase prediction accuracy is to make the global history register longer. This allows for more context in a prediction, reducing the effects of the aliasing problem. Figure 6 shows how NB predictors, with different bimodal counter lengths perform as the global history register length increases. It is clear to see that the misprediction rate decreases as the history register lengthens, although the rate of decrease reduces at higher history lengths.

7.2 Reactivity

The problem with the u-NB predictor from Section 3.1 is that it is insensitive to sudden probability distribution changes or program phase shifts. In contrast, saturating bimodal counters are able to react to such changes. The length of the bimodal counter determines its hysteresis, or how long it takes to react to changes.

We varied the length of the bimodal counters to find the optimum length. For the benchmarks given, the optimum performance was 4-NB, as shown in Figure 6. Longer counters cause degraded predictions since since they take too long to react to changes. Shorter counters cause degraded predictions since they forget reliable old history in favour of extremely transient behaviour. Note that some n-NB implementations outperform u-NB since u-NB is never able to ‘forget’ old history, so it becomes progressively more difficult to react to phase changes.

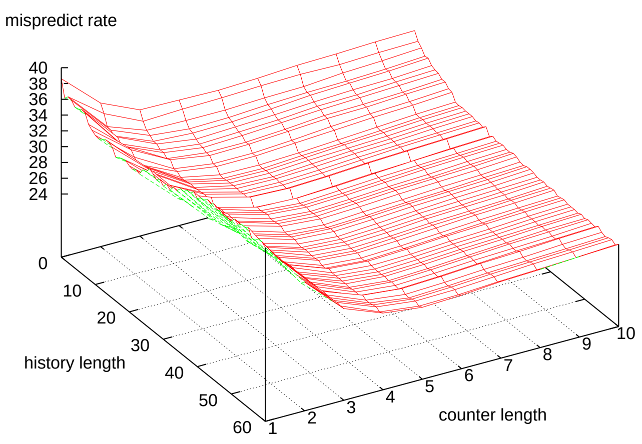


Fig. 6. Naive Bayes predictor accuracy varies with history length and counter length

7.3 Set Associativity

A common reason for low prediction accuracy is *aliasing*. The gshare predictor reduces aliasing by incorporating some low order bits from the branch instruction address into table lookup index. Another approach to reduce aliasing is *set associativity*. In this case, there are several NB predictors operating in parallel (NB/sa). The appropriate NB predictor to use for branch instruction b is based on the low-order bits from the branch instruction address. This approach is commonly used in processor caches. We investigate how it works for NB predictors. The main drawback is the growth in storage requirements for the predictor. The storage space grows exponentially with the number of instruction address bits used.

Figure 7 shows how set associativity affects the performance of the 2-NB and 4-NB predictors, each with a global history register length of 20. The x axis shows the number of bits of branch instruction address used, so the number of NB predictors will be 2^x. The graph shows how NB/sa prediction accuracy increases with set associativity. This is due to the decreasing amount of aliasing. However once the set-associativity exceeds a certain amount then the accuracy begins to degrade, presumably because there are so many parallel n-NB predictors, each for so few branch instructions that there is little or no global correlation between branch instructions.

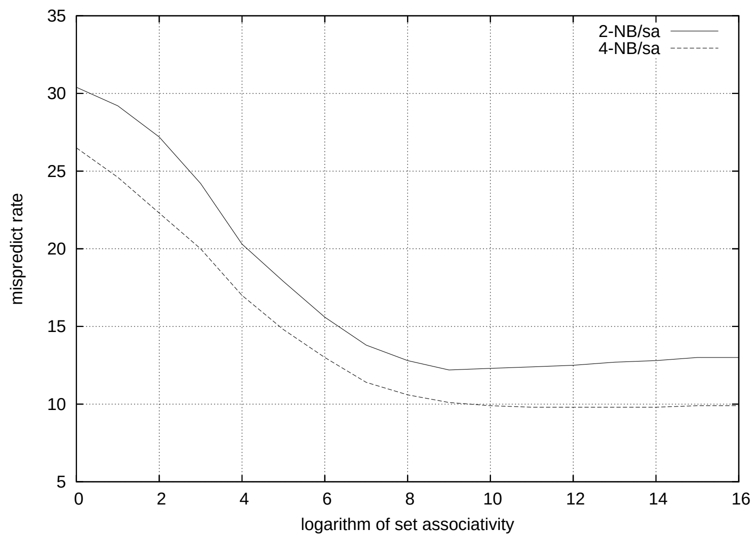


Fig. 7. Predictor performance improves with set-associativity

7.4 Network Connectivity

Section 2.2 described how the performance of the NB predictor can be improved by adding augmenting dependence edges between the attribute nodes, resulting in an augmented Naive Bayes predictor (aNB). This section investigates how these extra edges affect prediction accuracy. The conditional probability terms simply gain extra dependent variables. So for NB each term has the form $P(X_i|C)$ which translates into two counters, for when $(X_i = 1, C = 0)$ and when $(X_i = 1, C = 1)$. For TAN, each term has the form $P(X_i|X_{i-1}, C)$ which translates into four counters, for the four possible combinations of values for (X_{i-1}, C). In the general case, if each term has m dependent variables, then it requires 2^m counters.

The update scheme only examines $n+1$ counters each time, one for P(C) and one for each of the n history bits. It uses the values of the dependent variables to determine which counter to select for each term. Similarly, a likelihood check for a particular outcome only examines the top bits of $n+1$ counters.

Figure 8 shows how increased network connectivity affects the performance of the 2-aNB and 4-aNB predictors, each with a global history register length of 20. The x axis represents the *maximum* number of augmenting edges (AEs) per node. (Edges always point forwards, so X_1 can only depend on C, X_2 on C and X_1 whereas X_n can depend on C and all of $X_1, X_2, \ldots, X_{n-1}$.)

When the number of AEs is 0, the predictor is the special case of NB. When the number of AEs is 1, the predictor is the special case of TAN.

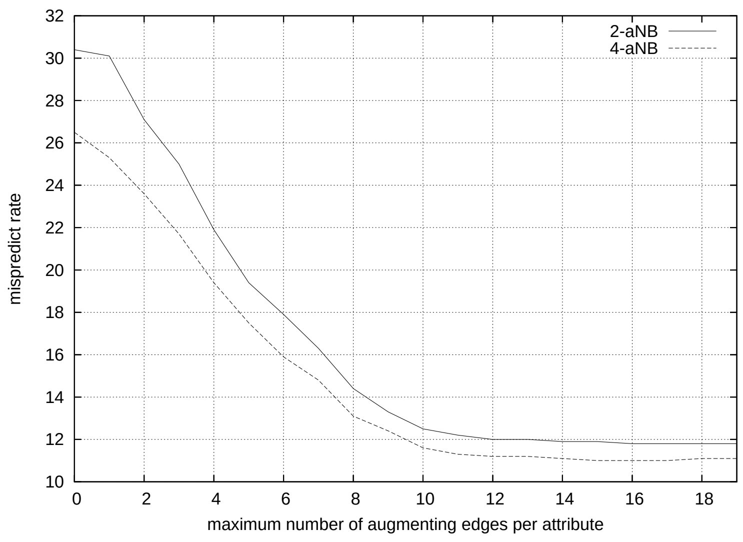


Fig. 8. Performance improves with increasing numbers of augmenting edges

8 Predictor Comparison: Improved NB versus gshare

Whereas Section 6 compared the performance of the n-NB predictor with equivalently sized gshare predictor, this section compares the improved version of the NB predictor with gshare. We combine all the NB performance enhancements from Section 7. Note that some enhancements seem to provide greater improvements than others. A more detailed empirical investigation would measure the ratio of accuracy improvement to storage overhead for the different enhancement schemes. Again, it is not clear how the various enhancements interact with one another. For simplicity, here we assume independence. A more detailed study would investigate this empirically!

We conducted numerous experiments, although we did not exhaustively explore the parameter space. Disappointingly none of our parameter settings enabled the bayesian predictor to achieve better accuracy than gshare, although many settings were close. For instance, given a aNB/sa predictor that uses 4-bit counters, 31 bit history, maximum of 15 augmenting edges and 256-way set associativity, the average mispredict rate is 4.7%. In contrast, an equivalently sized gshare predictor would have a history length of 19 bits, achieving accuracy score 4.0%.

9 Related Work

Several papers characterize branch prediction as a ML problem. For instance, Calder et al [11] use *decision trees* to predict branch outcomes at compile time, based on static program features. Fern and Given [12] formulate dynamic branch prediction as an online learning problem. They use *ensemble learning* techniques that are suitable for 'resource-limited environments.' However they do not provide hardware implementation details and they focus on a small set of branches that are difficult to predict.

Vintan [13] pioneers the idea of using *perceptrons* for branch prediction. Jimenez and Lin [14, 15] go on to show how perceptrons can be implemented in realistic hardware, and achieve better results than existing non-ML predictors. However they require complex techniques to disguise the latency of their perceptron predictor, involving cascaded predictors and pipelining.

Yeh and Patt [5] study a parameterized family of two-level adaptive predictors. They fully explore this small parameter space, which has 9 members. Their principled approach is a good guide for our work. Emer and Gloy [16] devise a language to describe conventional branch predictor models, and then use *genetic programming* to evolve new predictor models. However they admit that the auto-generated predictors are 'logically complex and probably not directly implementable.'

Online feature selection [17] is another interesting ML problem. In our case, we could chose which augmenting edges to insert dynamically, and perhaps adapt for programs with different branching characteristics.

10 Concluding Remarks

This paper has shown that Bayesian Networks provide a useful formalism for describing a family of branch predictors. The common GAg and gshare schemes can be explained in relation to BNs in terms of conditional probabilities. We have sketched a potential hardware implementation of this ML technique, and performed some initial evaluation.

One interesting observation is that a full floating-point implementation of NB is outperformed by our discrete approximation, using 4-bit bimodal counters.

Although we have not yet fully explored the space, we have found BN predictors that approach the gshare accuracy (to within 1%). We believe that this framework provides promise for future branch prediction technology, particularly in terms of storage overhead reduction. Our current research is focusing on further exploration of the space, and application of ML methods to automatically *learn* the best network connectivity, while the predictor is in use.

It should be noted that in adopting the BN formalism, we are addressing a subtle aspect of branch predictors that has not been previously considered. The Machine Learning literature can be broadly divided into two camps—*discriminative* and *generative* learning. The form of learning we have used is generative, since we model the joint distribution $P(X, C)$. Neural branch predictors [14] are discriminative, since they model the posterior distribution $P(C|X)$. The exact advantages of each in any given situation are an area of active research and therefore constitute a novel and promising technology for the architectural community to consider.

References

1. Hamerly, G., Perelman, E., Lau, J., Calder, B., Sherwood, T.: Using machine learning to guide architecture simulation. Journal of Machine Learning Research **7** (Feb 2006) 343–378
2. Barham, P., Isaacs, R., Mortier, R., Harris, T.: Learning communication patterns in Singularity. In: Proceedings of the First Workshop on Tackling Computer Systems Problems with Machine Learning Techniques. (2006)
3. Hennessy, J.L., Patterson, D.A.: Computer Architecture A Quantitative Approach. 3rd edn. Morgan Kaufmann (2003)
4. Jiménez, D.A., Keckler, S.W., Lin, C.: The impact of delay on the design of branch predictors. In: MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture. (2000) 67–76
5. Yeh, T.Y., Patt, Y.N.: A comparison of dynamic branch predictors that use two levels of branch history. In: Proceedings of the Annual Symposium on Computer Architecture. (1993) 257–266
6. Smith, J.E.: A study of branch prediction strategies. In: Proceedings of the Annual Symposium on Computer Architecture. (1981) 135–148
7. McFarling, S.: Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation (Jun 1993)
8. Friedman, N., Geiger, D., Goldszmidt, M.: Bayesian network classifiers. Machine Learning **29**(2-3) (1997) 131–163
9. Jiménez, D.A.: Second championship branch prediction (2006) `http://camino.rutgers.edu/cbp2/`.
10. Loh, G.H.: Simulation differences between academia and industry: A branch prediction case study. In: International Symposium on Performance Analysis of Software and Systems. (2005) 21–31
11. Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., Zorn, B.: Evidence-based static branch prediction using machine learning. ACM Transactions on Programming Languages and Systems **19**(1) (1997) 188–222
12. Fern, A., Givan, R.: Online ensemble learning: An empirical study. Machine Learning **53**(1-2) (2003) 71–109
13. Vintan, L., Iridon, M.: Towards a high performance neural branch predictor. In: International Joint Conference on Neural Networks. Volume 2. (1999) 868–873
14. Jiménez, D.A., Lin, C.: Neural methods for dynamic branch prediction. ACM Transactions on Computer Systems **20**(4) (2002) 369–397
15. Jiménez, D.A.: Improved latency and accuracy for neural branch prediction. ACM Transactions on Computer Systems **23**(2) (2005) 197–218
16. Emer, J., Gloy, N.: A language for describing predictors and its application to automatic synthesis. In: Proceedings of the 24th annual international symposium on Computer architecture. (1997) 304–314
17. Fern, A., Givan, R., Falsafi, B., Vijaykumar, T.: Dynamic feature selection for hardware prediction. Journal of Systems Architecture **52**(4) (2006) 213–224

Integrated CPU and L2 cache Frequency/Voltage Scaling using Supervised Learning

Cosmin Rusu, Nevine AbouGhazaleh, Alexandre Ferreira, Ruibin Xu,
Bruce Childers, Rami Melhem, and Daniel Mossé
{rusu, nevine, apf75, xruibin, childers, melhem, mosse}@cs.pitt.edu

Department of Computer Science, University of Pittsburgh

Abstract. Multiple clock domain (MCD) chip design addresses the problem of the increasing clock skew in the different chip units. MCD design opens the opportunity for independent power management in each domain when used in conjunction with dynamic voltage scaling (DVS). A significant power and energy improvement has been shown for finer control of each domain voltage rather than managing the chips single voltage, as in traditional chips with global DVS. However, published policies in the literature focus on each domain in isolation without considering the possible inter-domain effects when varying their clock/voltage from other domain.

In this paper we propose to use a supervised machine learning technique to automatically derive an integrated CPU-core and on-chip L2-cache DVS policy. Our policy relies on simple performance counters that can be easily monitored. We discuss the machine learning process and the implementation issues associated with our technique. We show that our derived policy improves on traditional power management techniques used in MCD chips. Our technique saves up to 34% (10% on average) over a DVS techniques that apply independent DVS decisions in each domain. Moreover, energy and energy-delay product results are within 3% of a near-optimal scheme.

1 Introduction

Dynamic Voltage Scaling (DVS) is a technique that can be used to reduce power consumption in CMOS digital circuits. A lower frequency of operation gives the possibility that a lower supply voltage can be applied. A convex relationship holds between frequency and power consumption for specific types of circuits and thus a small decrease of frequency/voltage can have a substantial impact on energy [14].

Due to the continuous increase in the number of transistors and lower feature size, higher chip densities create a problem for clock synchronization among different chip computational units. An effective solution to this problem is the use of design techniques for multiple clock domains (MCD) chips. In MCD, a processor chip is divided into multiple domains. Each domain operates synchronously with its own clock, and communicates with other domains asynchronously through FIFO queues. MCD design allows for fine grain power management of each domain especially when using dynamic voltage and frequency scaling (DVS). Since each domain has its own clock and voltage (i.e., independent of the other domains), DVS can be applied in each domain for an extra level of power management (rather than applying DVS at the chip level). Power and energy can be reduced with minimal impact on performance by dynamically reducing the clock speed and voltage in domains with low activity.

Several power management policies have been proposed to incorporate DVS into MCD chips. The published results show a significant power and energy improvement over applying DVS to a fully synchronized chip (i.e., with a single master clock) [7]. However, these policies focus on each domain in isolation without considering the possible effect of varying one domain's clock speed and voltage on other domains. Moreover, existing techniques rely on online heuristics.

In this work, we are interested in minimizing the overall energy or energy-delay product in a processor. We are especially interested in the CPU-core and the on-chip L2-cache, as they consume a large fraction of the total power in current processors. In this paper, we propose a novel methodology to derive an integrated CPU-core and L2-cache DVS policy. The integrated policy identifies application phases at runtime and takes corresponding actions (i.e., setting the voltage and frequency of both the processor and the L2-cache). The policy is derived with a supervised learning process on a representative training workload. We present and evaluate a policy that optimizes for either energy or energy-delay product of the entire processor (including the core and caches).

The rest of the paper is organized as follows. We briefly discuss related work in Section 2. Our problem description is given in Section 3. We describe the supervised learning technique we use to determine an integrated CPU-core and L2-cache DVS policy in Section 4, followed by evaluation in Section 5. Finally, we conclude the paper and discuss future work in Section 6.

2 Related Work

DVS was extensively explored for a variety of systems (from embedded devices to server farms) and application areas. For embedded systems, DVS techniques save energy by lowering the voltage and frequency for just-in-time completion of real-time applications [9, 3, 14]. For personal computers running Linux, DVS is used to lower the energy consumption while maintaining performance requirements of applications and good responsiveness of interactive jobs [5]. For web servers, utility-based DVS schemes adapt the frequency and voltage according to the incoming load [1]. In server clusters, DVS is used as a local power management scheme aware of Quality-of-Service constraints [11]

Multiple clock domains (MCD) are proposed as a fine grain processor DVS mechanism in [7]. Magklis et al. propose an online power management policy that monitors queue occupancy of a domain and adapts the domain's voltage accordingly [8]. For each domain, the policy computes the change in the average queue length among consecutive intervals. When queue length increases, the voltage and clock speed are increased. Similarly, when queue length decreases, the voltage and clock speed are decreased. However, this policy does not take into account the cascading effects of changing a domain voltage on other domains. Another technique by Magklis et al. uses a profile-based approach to identify program regions that justify reconfiguration [7]. This approach incurs extra overhead due to profiling and analysis phases for each application under consideration. In contrast, our technique learns the DVS policy with training samples and can be directly applied to new applications without profiling. Zhu et al present architectural optimizations for improving power and reducing complexity [17]. Voltage scaling of off-chip L2 caches for embedded systems is studied in [10].

Sherwood et al. showed that programs have repeatable phase-based run-time behavior over many hardware metrics, such as cache behavior or branch prediction [13]. The authors also provide a tool, called SimPoint, that automatically identifies and clusters the phases in a program in order to speed up architectural simulations [12]. Application phases and predictable behavior are essential to our work as well.

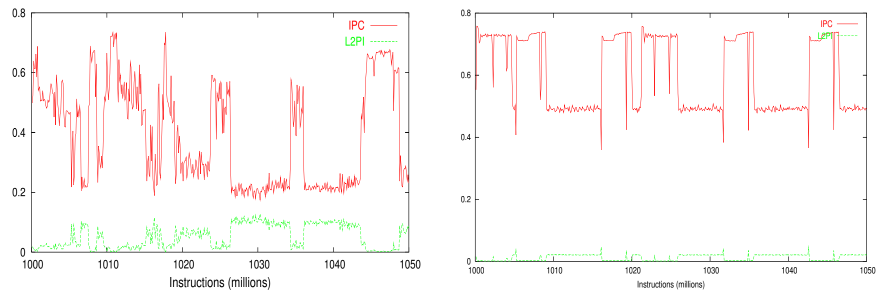
Applying machine learning techniques to reconfigure architectural and compiler settings is relatively a newly explored field. Wildstrom et al. present a policy to alter server configuration in reaction to workloads [15]. The policy learns to identify preferable CPU and memory configurations. They showed significant performance benefits using machine learning policy over any fixed configuration. Cavazos et al. use supervised learning to predict which application's basic blocks can benefit from scheduling [2]. The learned policy selects whether to schedule a block or not. The policy achieves most of the potential performance improvement with significantly less overhead.

3 Problem Description

A typical application goes through phases throughout its execution. An application has varying cache/memory access patterns and CPU stall patterns. In general, application phases correspond to loops, and a new phase is entered when control branches to a different code section. Since we are interested in the performance and energy of the CPU-core and L2-cache, we characterize each code segment in a program with two metrics: cycles per instruction (CPI) and number of L2 accesses per instruction (L2PI). CPI and L2PI are selected as indications of the amount of workload in the CPU-core and L2-cache, respectively. Examples of CPI and L2PI showing different program phases can be seen in Figure 1 for two benchmarks: *gcc* and *gzip* (from the SPEC2000 benchmark suite).

Intuitively, each program phase has a different requirement and preference toward a certain "configuration" of the CPU-core and L2-cache frequencies. For example, if a section of code is CPU bound, it will benefit from running at high CPU frequencies, and may be insensitive to L2-cache latency. On the other hand, a memory bound phase benefits the most from reducing the gap between the core and cache performance. This is precisely the intuition behind our approach. Our goal is to construct an integrated CPU-core and L2-cache DVS policy that identifies application phases and selects good frequencies for the CPU-core and L2-cache domains for each section of code.

Clearly, the L2-cache and CPU frequencies can be set independently based on activity represented by CPI and L2PI. Thus, we need to answer the following questions about an integrated policy: (a) Is an integrated CPU-core and L2-cache DVS scheme better than an independent scheme? (b) What are the mechanisms to be adopted with



(a) *gcc* phases (b) *gzip* phases

Fig. 1. Application phases variation throughout execution.

respect to these options? (c) What are the frequencies/voltages to be chosen at each program phase? We attempt to answer these questions throughout this paper.

One approach to building an integrated DVS policy is inspired by control systems. A phase change can be easily identified from simple performance counters. For example, a decrease in the CPI (after filtering noise) may suggest that a higher CPU frequency is needed, or that a higher cache latency is tolerable. A stable system phase is defined as a small variation (within a threshold) around the average CPI. The goal is then to change (i.e., increase/decrease) the CPU and/or L2-cache frequencies when a phase shift is identified.

The problem with a control approach is not identifying application phases, but selecting the correct frequencies on phase changes. The problem is that we can identify the correct action *towards* the optimal configuration, but not the optimal configuration itself. Using performance counters, we could decide, for example, whether the frequency of the CPU should be increased or decreased, but not the exact amount. This is because phase changes are not gradual, but *instantaneous*, corresponding to a jump to a different section of code. As soon as the jump is taken and the code enters a new phase (with stable CPI) there is no more feedback regarding how good the frequency change actually was, and if it was just a step in the right direction. Typically, there is little correlation between the amount of variation of some performance metric (such as CPI) and the right frequency. Furthermore, for more complex metrics such as energy or energy-delay product, even the step towards the correct action (i.e., increase or decrease frequency by one level) is hard to identify, as it is not trivial to estimate how energy consumption relates to the performance counters.

4 An Integrated CPU-core and L2-cache DVS policy

Because control-based approaches can fail to identify a good policy for integrated CPU-core and L2-cache DVS, we propose an approach based on a supervised machine learning. Our technique derives a policy expressed in the form of propositional rules for a particular system by analyzing a training program workload. For a given architecture, our approach analyzes the system to derive a DVS policy for both the CPU-core and L2-cache to optimize the energy-delay product. The approach describes the *state* of the system under different program behaviors and run-time system characteristics. A program behavior description captures the instruction level parallelism and cache demands of the application and a run-time characteristics description captures program latencies during a given program phase. The goal is to identify for each possible system state the correct *action*. An action determines how the CPU-core and L2-cache frequencies should be adjusted to minimize energy-delay product. The derived policy is thus a function that maps states to actions that take into account the effect on the energy and delay.

We first describe the methodology to obtain the training data used to learn the policy and then our learning approach.

4.1 Obtaining Training Data

It is our hypothesis that for a relatively simple (single issue) processor the system state that encapsulates the program behavior can be described by simple performance metrics. These metrics are the CPI and L2PI, which can be determined from hardware performance counters. The CPI indicates the CPU utilization; however, it does not by itself fully describe program phases. For example, a high CPI corresponds either to a high cache miss ratio, a high cache access latency, or long instruction latencies (such as division). Adding the L2PI into the state description eliminates the confusion and more fully describes application behavior. However, the L2PI does not take into account the effective latency of cache accesses, and to fully characterize the program, this latency has to be factored into the state description. We describe the effective access latency as a tuple of CPU-core and L2-cache frequencies. This representation of cache access latency provides similar information to the effective cache access latency but it also captures the energy, as energy cost is closely related to the operating frequencies.

Thus, a state is described by four parameters: CPI, L2PI, CPU-core frequency and L2-cache frequency. CPI and L2PI are continuous variables and need to be discretized. We choose a number of intervals (discretization bins) for both CPI and L2PI in such a way that the samples in the training data are distributed evenly. For example, because of the L2-cache efficiencies in current designs, if most samples have low L2PI, this would consequently create more L2PI ranges with lower values (i.e., finer granularity where the density is higher). Let K and L be the number of discrete values of the CPI and L2PI, respectively. Let M be the number of available CPU frequencies and N be the number of cache frequencies. The state is a table $State[CPI_k][L2PI_l][i][j]$, where CPI_k and $L2PI_l$ are the discretized values of CPI and L2PI ($0 \leq k < K$ and $0 \leq l < L$), respectively. i and j are the CPU-core and cache frequencies ($0 \leq i < M$

and $0 \leq j < N$), respectively. For each state we want to determine the action that minimizes energy-delay product.

The training data used to learn the policy is obtained from training benchmarks in the following manner. We run all training code at all CPU/cache frequency combinations (MN combinations). A sample is defined as a continuous sequence of code of fixed number of instructions equal to $size$. Thus, a set of training benchmarks with a total of $inst$ instructions and $size$ instructions will generate $C = inst/size$ code samples for one particular CPU/cache frequency, and MNC samples for all frequency combinations. We denote the samples by $S^c_{ij} = \{CPI^c_{ij}, L2PI^c_{ij}, ED^c_{ij}\}$, where c represents the code sample ($0 \leq c < C$). Each sample contains three values: CPI^c_{ij}, $L2PI^c_{ij}$, and ED^c_{ij}, namely, the discretized CPI, discretized L2PI, and energy-delay product of the sample while running at frequencies i and j.

After collecting these values for all samples, S^c_{ij}, the correct action for each state is determined as follows. Since for each section of code all the possible frequency combinations are available, the best action can be determined by adding the energy-delay product of each sample running at the new frequency. Since different sections of code may have the same state, an array that accumulates all values for the same state are used: $Cum[CPI_k][L2PI_l][i][j][x][y]$, where CPI_k, $L2PI_l$, i, and j are the state parameters and x and y are the new CPU and cache frequencies (that is, the action). For each training sample S^c_{ij} and each possible action x, y (x is the next CPU frequency, y is the next cache frequency), we update the arrays as follows:

$$Cum[CPI^c_{ij}][L2PI^c_{ij}][i][j][x][y] += ED^c_{xy} \tag{1}$$

Equation (1) accumulates the energy-delay product for all training samples and all possible actions. After updating the counters for all samples, the action for each state is the one that minimizes the actions. After updating the counters for all samples, the action for each state, $State[CPI_k][L2PI_l][i][j]$, is the frequencies $\langle x, y\rangle$ that minimizes $Cum[CPI_k][L2PI_l][i][j][x][y]$.

4.2 Learning DVS Policy

With the training data, we can use supervised learning to derive the DVS policy. There are many supervised learning techniques, including logistic classification, neural network, decision tree, and propositional rule. We prefer the propositional rule approach because it is more compact, more expressive, and more human readable than the other techniques. Furthermore, propositional rules are easy to implement in hardware. In fact, we tried all the aforementioned techniques on the training data and the propositional rule approach had the least error.

We use the Repeated Incremental Pruning to Produce Error Reduction (RIPPER) learner [4]. The RIPPER algorithm is known to achieve low error rates while being efficient on large data sets. RIPPER represents the collected states in the form of prepositional (if-then) rules. Each rule specifies the desirable CPU frequency and cache frequency for the next program interval based on the current state. The learner is based on the Incremental Reduced Error learning IREP algorithm [6]. RIPPER repeatedly calls IREP to construct the rule set with low error rates.

IREP iteratively builds its rule set in a greedy fashion; one rule at a time. IREP works in two phases: growing and pruning phases. First, it randomly partitions the data set in to two subsets: growing and pruning sets. The rule growth phase constructs an initial rule set. It starts with an empty clause and then repeatedly adds sub-conditions to the antecedent. The sub-conditions maximize the coverage of the rule (represents more states). The stopping criterion for adding sub-conditions is either covering all the input states or not being able to improve the rule coverage. After growing a rule, the rule is immediately pruned in the pruning phase. Pruning is an attempt to prevent the rules from being too specific. IREP chooses the candidate literals for pruning based on a score which is applied to all the sub-conditions of the antecedent and evaluate the score using the pruning data. This process is repeated until all states are covered or the learned rules have very small error.

The resulting rules are generated in the form of: IF $<$*condition*$>$ THEN $<$*set freq*$>$, where *condition* is a conjunction of one or more of the following sub-conditions. $(CPI_{cur} \leq CPI_k)$, $(CPI_{cur} \geq CPI_k)$, $(L2PI_{cur} \leq L2PI_l)$, $(L2PI_{cur} \geq L2PI_l)$, $(c_f = i)$, and $(m_f = j)$ where CPI_{cur}, $L2PI_{cur}$, c_f and m_f are the current CPI, L2PU, CPU frequency and and cache frequency, respectively. *set freq* specifies the value of the next CPU or cache frequencies.

5 Evaluation

In this section, we evaluate the effectiveness of an integrated CPU-core and L2-cache DVS policy derived with the supervised learning technique from Section 4. We compare the derived policy to (a) a local clairvoyant solution, which is near optimal for the energy-delay metric and (b) an independent CPU-core and L2-cache DVS policy [8].

5.1 Experimental Setup

We use the Simplescalar and Wattch architectural simulators with an MCD processor extension [17]. The MCD extension by Zhu et al. models inter-domain synchronization events and voltage scaling overheads. We alter the design in [17] to merge their individual core domains into a single domain and to separate the L2-cache into its own domain. The simulated frequencies for both domains vary from 250MHz to 1GHz with 250MHz steps. Voltage scales linearly with the frequency in the specified range. Memory is considered an external domain with a fixed latency.

We evaluate the policy learned with our method using an Alpha-like core configuration. We use a small number of functional units and narrow decode/issue widths to emphasize the CPU-core and L2-cache performance gap. Wider issue and decode widths combined with more functional units increase ILP are more likely to mask cache latencies. The processor configuration used in our simulations is listed in Table 1.

To obtain the propositional rules, we use *JRip* from the WEKA data mining software package [16]. *JRip* is an optimized implementation of the RIPPER learner. The rules are produced based on the data collected for the given architectural configuration. Each rule specifies the desirable CPU frequency and cache frequency for the next program interval based on the current state: CPI, L2PI, old CPU and cache frequencies.

Table 1. Simulation configurations

Parameter	Configuration
Dec./Iss. Width	1/1
dL1 cache	64KB, 2-way
iL1 cache	64KB, 2-way
L2 Cache	1MB DM
L1 lat.	2 cycles
L2 lat.	12 cycles
Int ALUs	2+1 mult/div
FP ALUs	1+1 mult/div
INT Issue Queue	4 entries
FP Issue Queue	4 entries
LS Queue	8
Reorder Buffer	40

An important aspect of using JRip in the WEKA engine is the format of the training data, which affects the quality of the generated rule set. Although all the state parameters of the training data are discrete (cache and CPU frequencies are discrete in nature, while L2PI and CPI are discretized into bins), we specify in the input to JRip that all parameters are continuous to get a more compact rule set. Using JRip also involves tuning the parameters for the RIPPER algorithm. For instance, the RIPPER algorithm needs to partition the training data into a growing set and a pruning set. We choose the partition size to be two thirds for the growing set. Since RIPPER is a randomized algorithm, different randomization seeds will lead to different results. We experimented with different values and chose a seed value that reduced the error rate and rule set size for our input.

We run a mixture of integer and floating point benchmarks from SPEC2000. The simulations are split into "training" and "evaluation" data. The training data contains the samples used for deriving the policy (i.e., the mapping of states to actions). The policy is evaluated on the evaluation data. In particular, for SPEC benchmarks, the "train" input data set was used for training samples and the "ref input data set" was used for evaluation runs. For both training and evaluation simulations, we fast forwarded the first one billion instructions and simulated the following 500M instruction.

We normalize results to a clairvoyant technique. The clairvoyant policy is obtained by selecting the best CPU-core and L2-cache frequencies for each sample (that is, the CPU-core and L2-cache frequency combination that minimizes the metric). While the clairvoyant algorithm is optimal for energy, note that it is only an approximation of optimal when the metric is the energy-delay product, as minimizing the energy-delay product for every interval does not necessarily minimize the overall energy-delay product for the entire application. We refer to this technique as *local-clairvoyant* in case of optimizing energy-delay product and as *clairvoyant* when optimizing for energy. We report how far the optimized metric is from the local-clairvoyant and clairvoyant results.

We compare our derived policy versus a base policy proposed in [8]. The base policy periodically monitors CPI and L2PI to control the CPU-core and L2-cache domains independently. We use a 500K cycle control period for the periodic voltage changes.

5.2 Experimental Results

Using the methodology from Section 4.1, we derived an integrated DVS policy for our experimental target system. Figure 5.2 compares the energy-delay product resulting from using the independent DVS policy versus our integrated DVS policy. Data is normalized to a local-clairvoyant policy. Lower values in Figure 5.2 are better as they are closer to the local-clairvoyant results. In all applications, we achieve an energy-delay product lower than the independent DVS policy. Reduction in energy-delay product over the independent policy is up to 34% in *art* (10% on average) across all application. More interestingly, the energy-delay results from our policy is within 3% of the local-clairvoyant technique.

In this setting we divided the CPI values into 11 bins (discretization intervals), and eight L2PI bins. Data from the training phase were able to cover 945 states out of 1408 possible states (11 CPI bins x 8 L2PI bins x 16 frequency combinations).

Mapping the states table into rules using *JRip* involves an approximation error. The error rate obtained in our set of rules is 6%. This corresponds to coverage of the training data by the rules of 94% . This implies that the rules are a good approximation of the full training data. For the states not covered by the rules, the action selected, though different, is close to the original. In fact, the differences in the optimization metric results are so negligible that the average error (relative to the full table) across all benchmarks is just 0.1%.

From these results, we conclude that our learning methodology being aware of the CPU-core and L2-cache states is effective and able to derive beneficial policies for the optimization metric (energy-delay product) on our experimental platform.

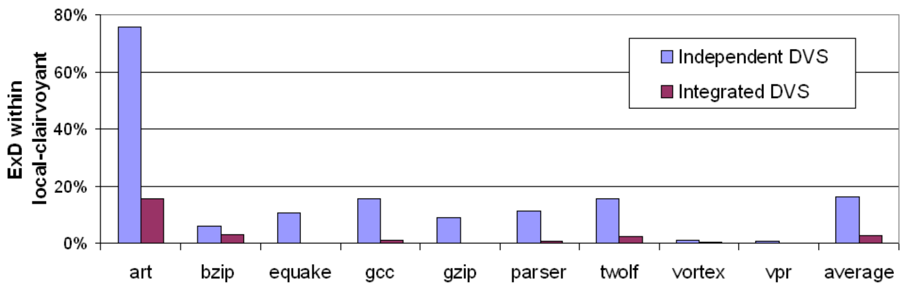


Fig. 2. Percentage increase in energy-delay relative to local-clairvoyant policy.

6 Conclusions and Future Work

In this work, we proposed the use of two important techniques for controlling the power and energy consumption in multiple clock domain processors. First, we proposed an integrated CPU-core and L2-cache DVS scheme that is based on simple performance counters (cache misses and instructions per cycle). Second, we used a supervised machine learning technique to derive a DVS policy for a given architecture. Our proposed scheme learns a frequency and voltage setting policy for scaling both CPU-core and L2-cache simultaneously. Our policy is within 3% of a locally clairvoyant policy.

In future work, we intend to study the impact of the different architectural configurations on our technique's accuracy. Also, we will investigate the significance of varying the learning process parameters (such as training data size, sampling size, and discretization granularity of both CPI and L2PI) on the results.

References

1. P. Bohrer, E. Elnozahy, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony. The case for power management in web servers. In *Power Aware Computing, Kluwer Academic Publications*, 2002.
2. John Cavazos, J. Eliot, and B. Moss. Inducing heuristics to decide whether to schedule. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 183–194, New York, NY, USA, 2004. ACM Press.
3. K. Choi, K. Dantu, W. Cheng, and M. Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *Proc. IEEE International Conference on Computer-Aided Design (ICCAD'02)*, San Jose, CA, November 2002.
4. W. W. Cohen. Fast effective rule induction. In *Proceedings of the 12th International Conference on Machine Learning*, June 1995.
5. K. Flautner and T. Mudge. Vertigo: automatic performance-setting for linux. In *Proceeding of the 5^{th} Symposium on Operating Systems Design and Implementation (OSDI'02)*, December 2002.
6. Johannes Furnkranz and Gerhard Widmer. Incremental reduced error pruning. In *International Conference on Machine Learning*, pages 70–77, 1994.
7. G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of the 30^{th} International Symposium on Computer Architecture (ISCA'03)*, June 2003.
8. Grigorios Magklis, Greg Semeraro, David H. Albonesi, Steven G . Dropsho, Sandhya Dwarkadas, and Michael L. Scott. Dynamic frequency and voltage scaling for a multiple-clock-domain micro processor. *IEEE Micro*, 23(6):62–68, 2003.
9. D. Mossé, H. Aydin, B. Childers, and R. Melhem. Compiler assisted dynamic power-aware scheduling for real-time applications. In *Workshop on Compilers and Operating Systems for Low Power (COLP'00)*, Philadelphia, PA, October 2000.
10. K. Puttaswamy, K. Choi, J. Park, V. J. Mooney III, A. Chatterjee, and P. Ellervee. System level power-performance trade-offs in embedded systems using voltage and frequency scaling of off-chip buses and memory. In *Proceedings of International Symposium on System Synthesis (ISSS'02)*, Kyoto, Japan, 2002.

11. C. Rusu, A. Ferreira, C. Scordino, A. Watson, R. Melhem, and D. Mossé. Energy-efficient real-time heterogeneous server clusters. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 418–427, San Jose, California, April 2006.
12. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.
13. T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30^{th} International Symposium on Computer Architecture (ISCA'03)*, June 2003.
14. M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *First Symposium on Operating Systems Design and Implementation*, pages 13–23, Monterey, California, 1994.
15. Jonathan Wildstrom, Emmett Witchel, and Raymond J. Mooney. Towards self-configuring hardware for distributed computer systems. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 241–249, Washington, DC, USA, 2005. IEEE Computer Society.
16. I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2005.
17. YongKang Zhu, David H. Albonesi, and A. Buyuktosunoglu. A high performance, energy efficient gals processor microarchitecture with reduced implementation complexity. In *IS-PASS'05: Proc Intl Symp on Performance Analysis of Systems and Software*, pages 42–53, 2005.