Rapidly Selecting Good Compiler Optimizations using Performance Counters

John Cavazos[1] Grigori Fursin[2] Felix Agakov[1] Edwin Bonilla[1]
Michael F.P. O'Boyle[1] Olivier Temam[2]

Members of HiPEAC
[1]Institute for Computing Systems Architecture (ICSA)
School of Informatics, University of Edinburgh, UK
[2]ALCHEMY Group, INRIA Futurs and LRI, Paris-Sud University, France

Abstract

Applying the right compiler optimizations to a particular program can have a significant impact on program performance. Due to the non-linear interaction of compiler optimizations, however, determining the best setting is non-trivial. There have been several proposed techniques that search the space of compiler options to find good solutions; however such approaches can be expensive. This paper proposes a different approach using performance counters as a means of determining good compiler optimization settings. This is achieved by learning a model off-line which can then be used to determine good settings for any new program. We show that such an approach outperforms the state-of-the-art and is two orders of magnitude faster on average. Furthermore, we show that our performance counter based approach outperforms techniques based on static code features. Finally, we show that such improvements are stable across varying input data sets. Using our technique we achieve a 10% improvement over the highest optimization setting of the commercial PathScale EKOPath 2.3.1 optimizing compiler on the SPEC benchmark suite on a recent AMD Athlon 64 3700+ platform in just three evaluations.

1 Introduction

Automatically selecting the best set of compiler optimizations for a particular program is a difficult task and there has been much previous work on automatically searching for the best optimization settings [13, 21, 22, 28]. This work is based on iteratively enabling certain optimization phases, running the compiled program and, based on its performance, deciding a new setting. In [21], a new algorithm called *combined elimination* (CE) was shown to outperform all previous search based techniques in finding good settings with considerably fewer evaluations than prior techniques.

However, these pure search or "orchestration" approaches do not use prior knowledge of the hardware, compiler, or program and instead try to obtain this knowledge online. Thus every time a new program is optimized, the system starts with no prior knowledge. In our experiments this means on average over 600 evaluations (compile + run) to tune an application. In contrast, the technique presented in this paper uses knowledge about a program's behavior to automatically select the best optimizations with as little as 3 program evaluations. Specifically, we use the performance counters collected from a few runs of the program as input to an automatically constructed model which outputs a probability distribution of good compiler optimizations to use. Using this dynamic knowledge of how a particular program runs on a particular platform, on a set of benchmark suites we are able to achieve the same performance or better as combined elimination, two orders of magnitude faster on average. Thus, obtaining knowledge a-priori (involving a *one-off-cost* at the factory), we can significantly speedup the searching of good optimization sequences for any new program. In contrast, ”pure search” techniques always obtain knowledge online and must do so for any new program they optimize.

Performance counters have been extensively used for performance analysis in explaining program behavior [9, 3]. One of the first papers to investigate how they could be used systematically to select optimizations [23] showed impressive performance gains. However, the heuristic used was manually developed over a 12 month period using detailed simulations. Furthermore, the optimizations selected were also implemented by hand. A small change in the architecture would potentially require the entire process to be repeated. In contrast, our scheme is entirely automatic. It uses machine learning to automatically build a model which maps performance counters to good optimization options without human intervention and is thus portable.

The use of learned models to guide the selection of optimizations has also received recent attention [1, 5, 26]. Stephenson *et al.* [26], for instance, use a genetic programming approach to automatically learn individual compiler optimizations, such as the register allocation spill heuristic, within the Trimaran compiler. In this paper, however, we consider the problem of determining the best settings for a large number of optimizations within a highly-tuned commercial compiler, PathScale EKOPAth 2.3.1 [24] whose performance is as good as or better than Intel 9.0. We show significant performance improvements over the highest optimization level for this compiler.

In our previous work [1], we used static code features to obtain good optimizations for new programs being compiled. The static features were used to find the most similar program from a set of previously explored programs. This was used for estimating a distribution of good sequences for the matching program, from which optimizations to apply for the new program were drawn. The idea is that optimizations which performed well on a “similar” (previously explored) program will work well for the new program being compiled. This approach works well on multimedia kernels on embedded processors, but as we show in Section 5.3 it performs poorly on general purpose applications. In fact, there is little or no performance improvement over the highest optimization level provided by the PathScale compiler. The main reason is that static code features, which essentially characterize local code constructs such as loops, provide a poor global characterization once aggregated over many such code sections. Furthermore, code features are a poor mechanism to describe the dynamic behavior of large control-flow intensive programs. We show that a performance counter based scheme overcomes these challenges.

Using performance counters to select good optimizations is attractive as it exploits knowledge of the program’s behavior without requiring knowledge of the programming language syntax. However, a potential difficulty with a performance counters based characterization is that they are based on a specific input data set. In Section 5.4, we show that the selected optimizations are robust across varying data sets. In other words, the performance counters generated for one run with a particular data set are sufficient for our model to predict a set of optimizations that work well over a large range of different data inputs.

This paper is organized as follows. The next section provides a motivating example showing that performance counters can be used to select good optimizations. Section 3 describes the performance counters used in this paper and how they can characterize program behavior. This section also includes a brief description of how we use a simple modelling technique, *logistic regression* [4], to automatically learn a global optimizing heuristic. Section 4 describes the experimental setup and is followed in Section 5 by the experimental results and their analysis. This includes a comparison between our scheme and both combined elimination and random selection. We also compare against a static feature based modelling approach and evaluate the impact of data sets. This is followed by a summary of related work and concluding remarks.

Compiler	Evaluations	Execution time	Speedup
-O0	1	40.2	1
-Ofast	1	32.3	1.24
CE	240	18.0	2.23
PC Model	3	17.2	2.33

Figure 3. The execution time and speedup over -O0 for the best compiler setting obtained using different schemes on `181.mcf`. The column labelled Evaluations shows the number of times the code must be run to achieve this level of performance by each scheme.

2 Motivation

The information obtained from performance counters is a compact summary of a program’s dynamic behavior. In particular, they summarize important aspects of a program’s performance, e.g., cache misses or floating point unit utilization. Our approach uses this information to automatically select compiler optimization settings likely to improve program performance.

This section looks at just one of the programs evaluated in this paper to illustrate how such performance counters can be used to select good compiler optimizations. As the performance counter values are related to actual program performance, they can be used by a modelling technique (described in the next section) to select good optimization settings. Our model examines performance counter values of a new program and, using prior knowledge from previously examined programs, determines the optimization setting most likely to result in a speedup and improved performance counter values.

Performance Counters Figure 1 illustrates the use of performance counters. This graph shows the performance counter values for the `181.mcf` benchmark from the SPEC

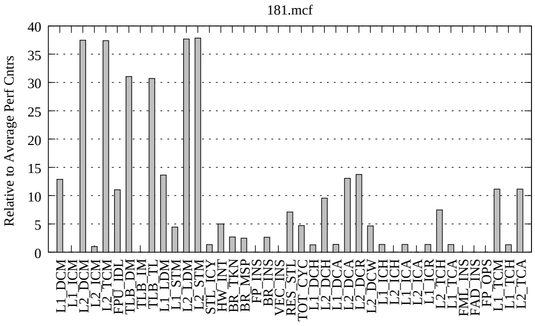


Figure 1. Performance counter values for `181.mcf` compiled with -O0 relative to the average values for the entire set of benchmark suite (SPECFP,SPECINT, MiBench, Polyhedron).

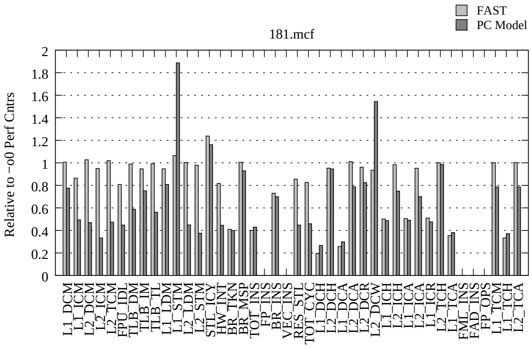


Figure 2. Performance counter values of -Ofast (FAST) and our scheme (PCModel) relative to -O0 for each performance counter for `181.mcf`.

benchmark suite on the AMD processor using the commercially available PathScale optimizing compiler. What is immediately apparent is that `181.mcf` is an unusual program - having a much greater number of memory access per instruction than average - up to 38 times more in the case of L2 store misses (L2_STM). A learned model should identify this and enable transformations that reduce the impact of cache accesses.

Figure 2 shows the performance counter values after applying two optimization schemes, -Ofast (FAST), the highest optimization setting available with PathScale and the setting found by our performance counter model (*PC-Model*). PCModel is able to significantly improve the use of the L1 and L2 cache. This is shown in the rightmost three bars of Figure 2 in the columns labelled L1_TCM (L1 total cache miss), L1_TCA (L1 total cache accesses) and L2_TCA (L2 total cache accesses). For instance, the model is able to reduce the number of L1 cache misses by 20% which has the effect of reducing the number of L2 accesses by 20%. -Ofast, on the other hand, has no effect on these values.

Performance Figure 3 shows the speedups obtained by -Ofast, CE, and PCModel over -O0. -Ofast is able to achieve a 1.24 speedup over -O0 while PCModel gives a speedup of 2.33, i.e., a speedup of 1.88 over -Ofast. It achieves this performance improvement with just 3 evaluations using a learned model trained offline. If we compare this to the performance of combined elimination (CE) [21], our scheme gives a slightly greater improvement. Furthermore, combined elimination requires 240 evaluations on this benchmark. Using our trained model, we are able to achieve greater performance improvement over the state-of-the-art, approaching two orders of magnitude fewer evaluations.

Transformations Examining the transformations selected by PCModel, it is apparent that locality enhancing loop optimizations have been enabled. In effect, the automatically generated model has learned that `181.mcf` has a problem with its memory usage and has selected transformations to overcome this. If, however, we examine the transformations selected by -Ofast and PCModel we see that they *both* enable the loop optimizer -LNO which is aimed at exploiting locality. On closer inspection, the major difference is that our model decides to turn on the -m32 flag, i.e., generate 32-bit code rather than the default 64-bit for the AMD. It does this because the number of data cache accesses and branch instructions are high. Figure 1 shows that the data cache accesses are relatively high (L1_DCM and L2_DCM) for this benchmark. Also, looking at the number of branch instructions (BR_INS) we see it is more than 2.5 times the average.

According to the AMD compiler manual [2] the -m32 option “can improve performance if your program has lots of variables of the type long and/or pointers. As these datatypes are 32-bit in x86, this switch will reduce the memory footprint of your program.” We note that -m32 is only useful for a few programs, and our model decides based on the characteristics (performance counters) of each program when it should be applied. Many manufacturers include the -m32 option in the SPEC ”peak” flags for some codes when using PathScale. By examining the code of `181.mcf` we see it accesses its main data structure through pointers in a loop. Also, it has a large number of branches executed proportional to the number of total instructions due to small tight loops. Reducing the pointer size, by using -m32, reduces the number of I-cache data misses dramatically for `181.mcf` as can be seen in Figure 2.

Thus, our model has learned that data cache misses and branch instructions (via the performance counter data) are the critical characteristics of this program and suggests the

Performance Counter Name	**Meaning**	**Average Values**
HW_INT	Hardware interrupts	0.000
RES_STL	Cycles stalled on any resource	0.660
STL_ICY	Cycles with no instruction issue	0.035
TOT_CYC	Total cycles	1.099
TOT_INS	Instructions completed	1.000
VEC_INS	Vector/SIMD instructions	0.017
Floating Point Instruction Statistics		
FAD_INS, FML_INS, FP_INS, FP_OPS, FPU_IDL	Floating point: Adds, Multiplies, Total Insns, Total Ops, Cycles Idle	0.030, 0.036, 0.066, 0.066, 0.473
Branch Instruction Statistics		
BR_INS, BR_MSP, BR_TKN	Branch instructions, Cond. Branches Mispredicted, Cond. Branches Taken	0.047, 0.002, 0.035
Level 1 Cache Statistics		
DCA, DCH, DCM	Data Cache: Accesses, Hits, Misses	0.475, 0.472, 0.004
ICA, ICH, ICM, ICR	Instruction Cache: Accesses, Hits, Misses, Reads	0.316, 0.315, 0.0006, 0.315
LDM, STM	Load Misses, Store Misses	0.0015, 0.0016
TCA, TCH, TCM	Total Cache: Accesses, Hits, Misses	0.789, 0.790, 0.004
Level 2 Cache Statistics		
DCA, DCH, DCM, DCR, DCW	Data Cache: Accesses, Hits, Misses, Reads, Writes	0.003, 0.003, 0.0005, 0.0015, 0.0016
ICA, ICH, ICM	Instruction Cache: Accesses, Hits, Misses	0.0006, 0.0006, 0.000002
LDM, STM	Load Misses, Store Misses	0.0004, 0.00008
TCA, TCH, TCM	Total Cache: Accesses, Hits, Misses	0.004, 0.003686, 0.0005
TLB Statistics		
TLB_DM	Data translation lookaside buffer misses	0.0002
TLB_IM	Instruction translation lookaside buffer misses	0.000001
TLB_TL	Total translation lookaside buffer misses	0.0002

Figure 4. Performance counters used. This table describes all the performance counters used in our study. The first column lists the performance counter acronyms, the second column gives a description, and the third column gives the average counter values normalized by total instructions executed across the entire set of benchmark suites.

compiler convert pointers from 64-bit to 32-bit, because 64-bit pointers are reducing the effective cache capacity and memory bandwidth. This demonstrates the strength of automatic model construction. It has no a priori human bias about what are important program characteristics or transformations – it learns solely based on empirical evidence.

This example show that performance counters can be used by a model to select optimizations that improve performance by examining the values of the counters. Furthermore, it can find good sequences rapidly. The next sections describe the performance counters used in this paper in greater detail and explains the technique used to automatically build an automatic optimizing heuristic.

3 Optimization selection based on performance counters

This section first looks at the performance counters used in this paper and illustrates that they can be used to characterize well-known properties of SPEC FP and MiBench. This is followed by a description of the modelling technique we use which is based on logistic regression. This is a standard machine learning technique which can learn whether an optimization is good or bad for a certain set of performance counter values and associates a probability with this decision.

3.1 Dynamic characterization of program behavior using performance counters

Modern processors are often equipped with a special set of registers that allow for measuring performance counter events with no disruption to the running program. These events can describe several characteristics of the running program, such as, cache hits and misses and branch prediction statistics. On the AMD Athlon, there are 4 registers for measuring performance counter events, but up to 60 different events can be measured. It is possible to collect anywhere between 4 and 60 types of events per run by multiplexing the use of the special registers. Since we aim at broadly characterizing the program behavior rather than studying a particular performance phenomenon, we have collected all 60 events. The performance counters used in this study are shown in Table 4. In order to use the collected statistics as inputs to our model, we normalized the value of each performance counter by TOT_INS, the total number of instructions executed. Normalizing the performance counters is important since it allows us to generalize across different benchmarks regardless of how long each benchmark

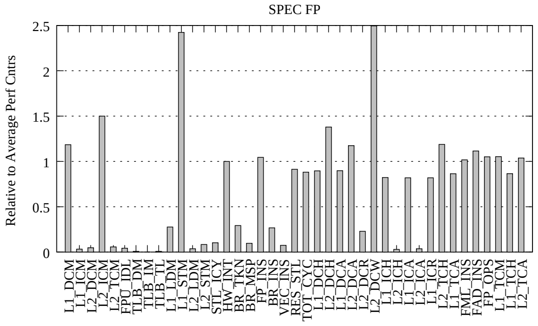


Figure 5. Performance counter values for SPEC FP average values for the entire benchmark suite compiled with -O0.

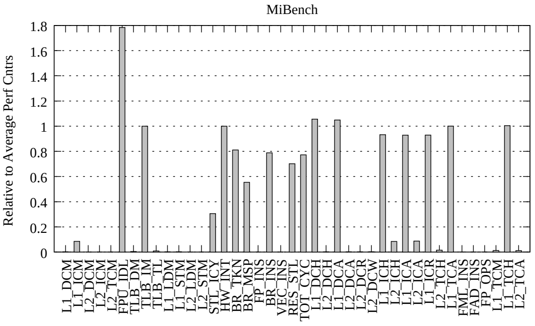


Figure 6. Performance counter values for MiBench average values for the entire benchmark suite compiled with -O0.

executes.

Table 4 also presents the average values for each counter across our benchmark suites. As can be seen, some counter events are relatively common, such as L1 data cache misses (L1_DCA) while others are relatively rare, such as instruction TLB misses (TLB_IM).

To see how the performance counters can characterize program behavior, we have examined two benchmark suites, SPEC FP and MiBench, out of the four benchmark suites we evaluated (SPEC INT, SPEC FP, MiBench, Polyhedron). We plotted the performance counter values for each of these suites relative to the average values for the entire collection of benchmarks. These values are shown in Figures 5 and 6 where the values are collected when compiled with -O0. It is interesting to see that MiBench exercises the cache hierarchy much less than SPEC FP which can be seen by the much larger number of L2 cache misses (L2_TCA) for SPEC FP. Also, MiBench has a relatively large number of branches compare to SPEC FP, almost twice as many, and many of the branches in MiBench are mispredicted, while more branches for SPEC FP codes are easier to predict (BR_MSP).

These results are not surprising and have been discussed in prior work. However, these graphs show that there are important aspects of program behavior that can be captured using performance counters. Furthermore, this information can serve as an input to a model that selects compiler optimizations. Using these statistics our model can learn to apply optimizations that will reduce the impact of cache misses for SPEC FP or branch misses for MiBench. The next section shows how we can use this information to select good optimizations automatically using machine learning.

3.2 Automatically learning a good model

The goal of model construction is to learn a mapping $\mathtt{x} \rightarrow \mathtt{t}$ between a set of performance counters features $\mathtt{x}$, and a set of good optimizations $\mathtt{t}$. Here $\mathtt{x}$ is a vector of the 60 normalized performance counters values, and $\mathtt{t}$ is a vector mask indicating which transformations are used in the sequence (i.e., each vector entry corresponds to a transformation, and $\mathtt{t}$ corresponds to a sequence of 0/1, 1 meaning the transformation is used). The goal of the model is to predict the best possible $\mathtt{t}$ for a program described by features $\mathtt{x}$. We emphasize this is not the phase-ordering problem. That is, our models do not find the best order in which to apply transformations. While this is an interesting problem, most compilers do not allow changing arbitrarily the order optimizations are applied.

In the following paragraphs, we first describe how the training set required for offline training is collected which is then used to construct a model. We then describe how this model is used and how sensitive its decisions are to different data inputs.

Training set

At first, a large number (500) of transformation sequences $\mathtt{t}$ are randomly sampled and the speedup for each of these sequences is recorded. Also, we require three additional runs of the program to collect the performance counter values $\mathtt{x}$. The runs where the speedup, relative to -Ofast, is smaller than 1 are filtered out, and the remaining data forms the training set. Then we use the standard *Leave One Out Cross-Validation* procedure for evaluating our models; therefore, the models are trained on $N-1$ benchmarks and tested on the Nth benchmark that has been left out. In our experiments, $N = 57$ and our models were trained with $56x500 = 28000$ training data points. The cost of obtaining

28000 training points is expensive, however it is one-off-cost incurred offline at the factory.

Model construction

The model is now built using the training set. The predictive modelling process is summarized in Figure 7. We use a probabilistic approach for predictive modelling, where we determine for each optimization t_i the probability p_i that it should be evaluated. The technique used is *logistic regression* [4]. Intuitively, it attempts to find the set of performance counter values for which enabling the transformation t_i leads to improved performance in the training set and also determines when disabling a transformation is preferable. In effect, it tries to draw a hyperplane in the multi-dimensional hardware counter space between those occasions when the transformation is best enabled and those occasions when it is best disabled. Borderline cases near the hyperplane have a probability, p, around 0.5 associated with them. Those which should be definitely enabled have $p >> 0.5$ and those which should be definitely disabled have $p << 0.5$. This is a standard machine learning technique and is computationally inexpensive - see [4] for a more detailed description. Note that gathering training data and construction of the model is an offline process, that is, it would take place "at the factor" before the compiler is shipped to the costumer.

Using the model

Given a new target benchmark, we first extract the performance counter features $\mathtt{x}$ by running the benchmark. This requires 3 runs of the benchmark. This features vector is then fed as input to our trained models which then outputs a probability p_i for each transformation $\mathtt{t}_i$ showing whether each transformation should be applied or not. We then sample from this probability distribution to generate a suitable compiler optimization setting. The program is then optimized based on the transformations selected and the new speedup is measured as shown in Figure 7. An advantage of this technique is that we can sample as many times as we wish to generate different settings. We later show in Section 5.1 that very few samples are required to achieve good performance. Furthermore, increasing the number of samples which are evaluated increases the performance obtained.

Data set sensitivity.

Predicting the best possible $\mathtt{t}$ for a program with features $\mathtt{x}$ is, in fact, not practical, since a program is usually run once on each data set. Therefore, when $\mathtt{t}$ is applied, the data set may in fact correspond to a features vector $\mathtt{x}'$ which is different from $\mathtt{x}$. However, in Section 5.4 we show that performance counter statistics are stable enough across data sets, i.e., transformation sequence $\mathtt{t}$ predicted using the feature vectors $\mathtt{x}$ of previous runs also performs well for new runs of the same program with different data sets.

4 Experimental Setup

This section briefly describes the experimental setup. First, the hardware platform, OS, and optimizing compiler are described. Second, we describe the benchmarks and the optimizations available for selection.

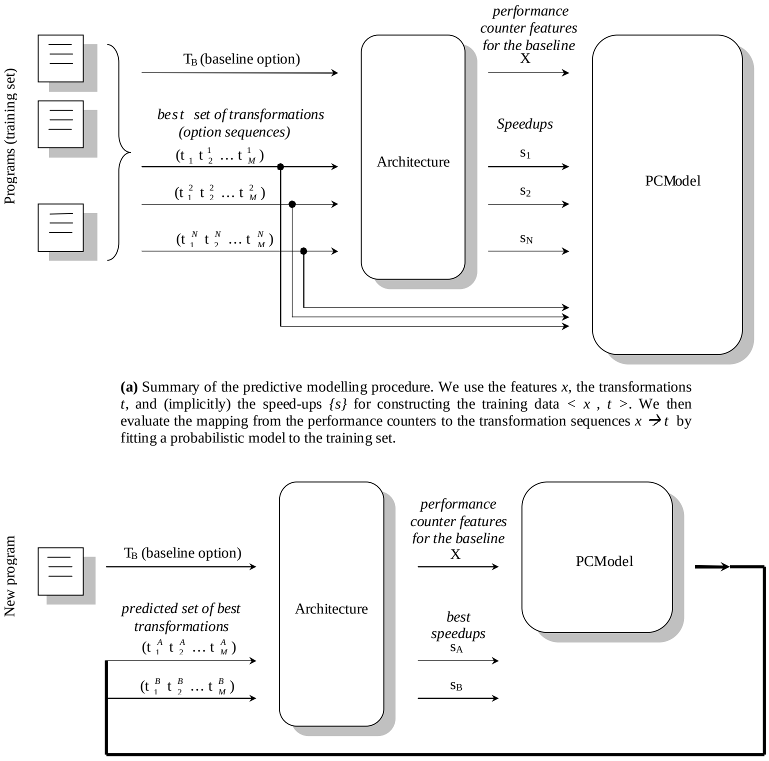
4.1 Platform

We perform 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 use the latest PAPI 3.2.1 hardware counter library [20] and PAPIEx 0.99rc2 tool to collect hardware performance counters for the benchmarks. Table 4 has a brief description of all the counters we use. PAPIEx works in the multiplexing mode allowing us to collect a large number of counters in one run. We collect performance counters using level -O0 so that characteristics of a benchmark are not masked by higher optimization levels (e.g., -Ofast). We use the latest open-source commercial PathScale EKOPath Compiler 2.3.1 [24] with the -Ofast flag, which we refer to as our *baseline*. This compiler is tuned to AMD processors and on average performs similarly or better than the Intel 9.0 compilers on the same platform.

In order to evaluate the stability of our measurements, we have executed multiple runs of each benchmark using `-Ofast` and have found there to be very little variance in the execution time, on average less than 0.3%.

4.2 Benchmarks

We evaluate our approach on widely-used benchmark suites written in C, C++, Fortran and Fortran 90. These are SPEC 95 FP (ref dataset), SPEC 2000 FP and INT (train dataset), Polyhedron 2005 [17], and MiBench [12]. We used train inputs for the SPEC 2000 benchmarks due to the large number of experiments we ran, on average over 1000 for each benchmark. SPEC and Polyhedron benchmarks are relatively large programs used for performance evaluation of servers and for comparison of performance of various compilers on these servers. These benchmarks are used by PathScale to tune their compiler suite. MiBench is a free, commercially representative embedded benchmark suite consisting of a large number of applications and kernels. We believe that all these programs cover a large variety of different dynamic behaviors. In the experimental



(a) Summary of the predictive modelling procedure. We use the features *x*, the transformations *t*, and (implicitly) the speed-ups *{s}* for constructing the training data < *x* , *t* >. We then evaluate the mapping from the performance counters to the transformation sequences *x* → *t* by fitting a probabilistic model to the training set.

(b) Inference using a predictive model. Given a new benchmark, we first extract performance counter features. These features are then fed into our trained models which then output a set of transformation sequences to apply to the new benchmark.

Figure 7. Training and using the models.

section, we partition the benchmarks into SPEC, which includes all *SPEC* benchmarks and *Others*, which includes MiBench and Polyhedron.

4.3 Transformations

The PathScale EKOPath compiler suite includes a PathOpt tool that randomly selects from a variety of global compiler settings. PathOpt iterates for a user-specified amount of evaluations and is used to iteratively find the best performance on a targeted platform and program. We select 121 flags that are known to influence performance and use PathOpt to apply 500 random settings of these flags on all benchmarks. We also allow our model to generate sequences with optimizations that are mutually exclusive, e.g, different unroll factors. When this happens, the compiler simply ignores all but the last option. We could have easily adapted our models to handle this directly (e.g., using a soft-max approach where the optimization with the highest probability is choosen), however this would make it harder to compare with combined elimination and random. The speedups for each setting were used along with performance hardware counters as training data for our logistic regression model.

5 Experimental results

In this section we evaluate our proposed technique in a number of ways. Initially, we report the performance improvement achieved by our technique compared to combined elimination and the number of evaluations each scheme needs to achieve such a level of performance. Since we use a probabilistic model we have generated our results for multiple trials. [1] This is then followed by a comparison of the performance of a number of different schemes with respect to the number of evaluations available. Here, we compare combined elimination (CE), our models (PCModel), and a random selection (RAND) approach which generates random transformation sequences. RAND is implemented using a pseudo-random number generator to choose which optimizations to apply in the sequence. Each optimization has a .5 probability of being used in each se-

[1] For random selection we have also generated our results for multiple trials.

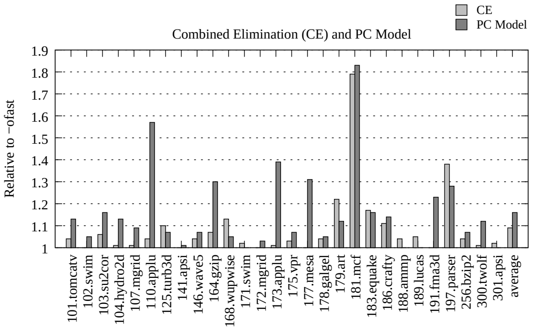


Figure 8. The speedup of Combined Elimination (CE) versus our model (PCModel) for the SPEC benchmarks relative to the performance of -Ofast. The number of evaluations used by PCModel is limited to 25 while CE uses on average 609.

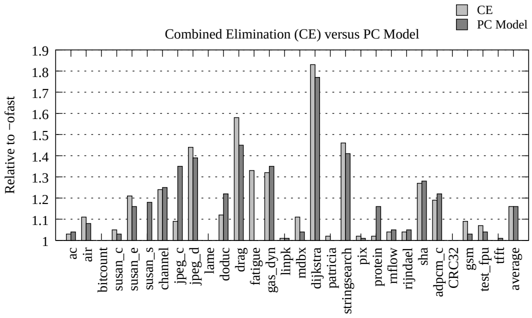


Figure 9. The speedup of Combined Elimination (CE) versus our model (PCModel) for the Other (non-SPEC) benchmarks relative to the performance of -Ofast. The number of evaluations used by PCModel is limited to 25 while CE uses on average 609.

quence. We then provide a study examining the use of static code features as a means of selecting compiler optimizations. Also, we examine how the optimizations selected by our model perform across different input data sets. Finally, we perform some analysis to evaluate the most important performance counter features. All results in this section, unless otherwise stated, are relative to -Ofast, the highest optimization level available in PathScale. Note, all of our techniques (CE,PCModel,Random) start with the -Ofast as their initial sample, so none of our techniques can perform worse than -Ofast. We emphasize that our models are built using leave-one-out cross validation, so the models are not trained using any information of the programs it is optimizing.

5.1 Speedup and number of evaluations for PCModel and CE

Figure 4.3 shows the speedups of our model and combined elimination relative to -Ofast on the SPEC benchmarks. PCModel achieves a speedup of 1.17 on average compared to 1.09 by CE. In this example, we have set the number of evaluations selected by our model to 25 evaluations. However, CE needs on average 609 evaluations of the program to achieve this. We note that both PCModel and CE can find significant improvements for several programs. For instance, PCModel finds improvements of 10% or more over -Ofast for half (14 of 28) of the SPEC benchmarks. We describe the best optimizations found for each program in a later section. Thus we are able to achieve better performance than CE with considerable fewer evaluations. In fact, our model achieves the same performance as CE on the SPEC benchmarks in only 3 iterations. Figure 4.3 shows a similar comparison for the other benchmarks. In this case both schemes give approximately the same performance improvement of 1.17. Figure 5 however, shows that CE needs a large number of evaluations to achieve this performance level, ranging form 240 evaluations up to 1550 with an average of 609 needed.

It is not suprising that the CE algorithm requires a large number of evaluations. The CE algorithm first evaluates the effect of each of the optimizations by turning them off one at a time. This is 121 evaluations for the entire set of optimizations we explored. Then it “combines” the knowledge gathered by these initial evaluations to choose optimizations that lead to better improvement when turned off. After several iterations of turning off single optimizations it converges to a setting where no additional flags turned off improve performance. We refer the reader to the combined elimination paper [21] for further details of the algorithm.

5.2 Performance versus number of evaluations for different schemes

To give a different view of how our model performs we considered the performance it achieves as a function of the number of evaluations we select from it. Figure 5.3 shows the performance achieved averaged across all the benchmarks versus the number of evaluations allowed. We also compare our approach relative to combined elimination and random selection. For each technique, the more evaluations allowed the better the performance achieved. Our model achieves the same performance after 60 evaluations as random does after 200 evaluations. Surprisingly, random selection does quite well! [2]

[2] Other papers [14, 1] have reported on the excellent performance of pure random search.

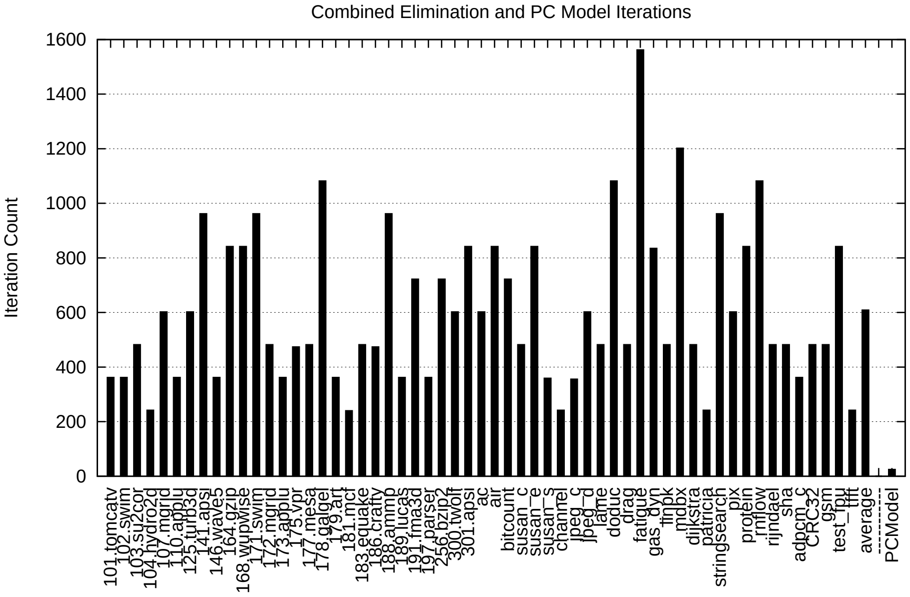


Figure 10. The number of evaluations of the CE algorithm per benchmark. The mininum evaluations is 240 (`181.mcf`), maximum is 1562 (`fatigue`), and the average is 609. We compared CE to PCModel with an evaluation count 25 (far right).

CE's initial iteration involves turning off each optimization in turn which gives a small speedup of 1.04 across the first 121 evaluations. Only after all 121 options have been evaluated does it improve its behavior by "combining" its results. However, after 200 evaluations it only achieves the same performance as that achieved by our model after 2 evaluations. Random selection achieves this in just 10 evaluations. RAND and CE both found an improvement of 17% over -Ofast using 200 and 60 iterations, respectively. CE finds less improvement at 12%. We note that for these experiments the CE algorithm was run to completion. On the other hand, given the nature of the RAND and PCModel algorithms, we could continue to construct sequences with these algorithms until we are satisfied with the optimizied performance of our application. Both, RAND and PCModel whould reach the same maximum available speedup in the limit, however PCModel should reach that speedup sooner.

We believe RAND and PCModel perform well in our experiments because the search space has many good points. However, in certain scenarios CE could outperform RAND (and models trained with RAND data). CE's main goal is to *eliminate* optimizations that degrade performance. If we encouter an optimization space with many optimizations that degrade performance of a benchmark, it would take a large number of iterations for RAND to construct a sequence with none (or few) of these degrading optimizations.

5.3 Static versus dynamic features

As mentioned in the introduction, a recent model-based approach [1] attempted to characterize programs using static code features. In this section, we quantitatively compare the merit of static (code) versus dynamic (performance counter) features using the same number of evaluations. Agakov *et al.* [1] applied their approach to the UTDSP, which are small embedded kernels containing often a single loop nest with affine array accesses. In order to extract the static features we also constructed features analogous to those used by Agakov *et al.* We extracted code features for several of the SPEC INT benchmarks, which are large control-flow intensive programs. We also used a K-nearest neighbor approach and build IID distributions similar to what was used by Agakov *et al.* to be as fair as possible. We choose neighbors using either code features or performance counter features and drew samples from the neighbor's IID distribution to apply to a new program.

The list of benchmarks used for the comparison is indicated in Figure 5.3. The static based approach finds some improvement in 4 of the 7 programs achieving an average speedups of just 1.01 when compared to -Ofast. Our performance counter model significantly improves over the static feature-based approach giving a speedup average of 1.08 over -Ofast and is therefore significantly better for characterizing programs with complex control flow, e.g.,

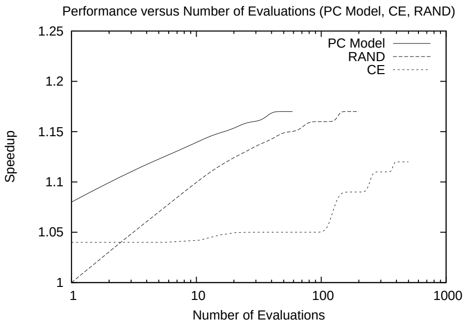


Figure 11. The speedup of Combined Elimination (CE), PCModel, and random selection averaged across all benchmarks versus the number of program evaluations used.

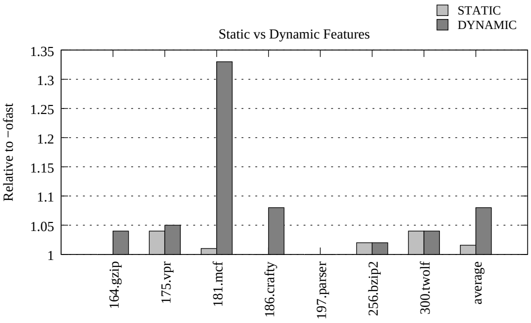


Figure 12. Performance of SPEC INT 2000 Benchmarks using static code features and dynamic features.

automotive_susan_c	1.03 ± 0.01	network_dijkstra	1.66 ± 0.01
automotive_susan_e	1.13 ± 0.04	network_patricia	1.00 ± 0.00
automotive_susan_s	1.08 ± 0.00	security_rijndael	1.01 ± 0.05
consumer_jpeg_c	1.07 ± 0.01	security_sha	1.18 ± 0.01
consumer_jpeg_d	1.16 ± 0.07	telecom_adpcm_c	1.06 ± 0.04
consumer_lame	1.00 ± 0.00	telecom_CRC32	1.66 ± 0.04

Table 1. Average performance speed-ups (and one standard deviation) for the performance of the best sequence of our models (PCModel) across different inputs.

`181.mcf` and `186.crafty`.

5.4 Data set sensitivity

In order to further explore the practical application of our approach under real conditions, we have evaluated its sensitivity to data set. In certain contexts, such as embedded processing, it is possible to execute/train a program many times on the same data set at design time because the data set will not vary much. The same goes for some scientific applications where matrix dimensions are the most critical data set parameters performance-wise but do not always vary, unlike the content of the matrices itself.

Still, in many other contexts, a program uses different data sets during each of its runs. Therefore, for a subset of our benchmarks (several MiBench applications), we collected 20 data sets per benchmark, and we evaluated a scenario where the best sequence predicted from our model (PCModel) using performance counters collected from one data set is used to compile a program which is then run with a variety different data sets. This scenario helps analyze whether the performance counter features vector is sufficiently stable across data sets. Table 5.3 compares the performance of the best sequence predicted with our model on one data set applied to several datasets. As can be seen, the predictions from our model are quite stable across different datasets and always lead to a performance over -Ofast. We used MiBench benchmarks for these experiments given the readily available data sets for these benchmarks. In future work, we will investigate whether these conclusions hold for more complex programs such as SPEC INT.

5.5 Analysis of the importance of the performance counters

The goal of this analysis is to understand which performance counters are most important for predicting good optimization sequences. The fundamental objective in this context is *mutual information* between a *subset* of the performance counters and good optimization sequences (for definition of mutual information, see e.g. [18]). Our goal was to maximize the mutual information for subsets of the retained features.

In general, it is intractable to compute the mutual information exactly, therefore approximations need to be considered. For our analysis we applied a novel subset-selection approach which greedily maximizes the Gaussian approximation of the mutual information. For training data containing one good optimization sequence per benchmark, we found that over 95% of the total information (if all features are retained) was typically contained in just 15 performance counters. In contrast, conventional approximations disregarding interactions between the inputs (e.g. [27]) would typically require twice as many features to retain the same amount of information, which in our case is not much better than random selection (see Figure 13 *left*). Interestingly, while the choice of the informative features generally depends on the training data, we found that there was a lot of overlap between the performance counters found for various good transformation sequences (typically with one or two unique features per training set). Figure 13 *right* shows informative performance counters for training sequences constructed with the CE algorithm.

6 Related Work

Parello *et al.* [23] presents a systematic, but manual iterative approach for program optimization using dynamic features. At each iteration, performance counters are used to identify a performance anomaly of a program and a set of program transformations is suggested to solve this problem. Then the transformed program is run again to detect further performance anomalies. The process is manual and can take several weeks per benchmark. Moreover, it cannot tackle more than one anomaly in one step and it does not provide a learning mechanism to correlate the dynamic behavior of the program with the performance of transformations across applications. Our technique is fully automated and can generate heuristics to predict good optimizations in seconds.

In the area of predictive modelling, Zhao *et al.* use manually constructed cost/benefit models to predict whether to apply PRE or LICM[33]. They achieve 1% to 2% improvement over always applying an optimization, but at a cost of greatly increasing compilation time (by up to 68%). Their models appear to be quite complicated and have to be manually constructed. Our models, on the other hand, are simple and automatically constructed using machine learning.

During the past several years, the benefits of iterative compilation have been widely reported [14, 7, 8, 10, 13]. Iterative compilation is able to find optimization sequences that out-perform the highest optimization settings in commercial compilers and when applied to library subroutines they find solutions that compare favorably with highly-optimized hand-tuned vendor libraries [30, 11, 25, 29]. However, iterative compilation requires searching a combinatorially large space defined by the optimizations of interest. This search can take several days to weeks depending on the running time of the program, speed of the compiler and target architecture, and thoroughness of the search. There have been a number of papers focusing on reducing the cost of iterative optimization.

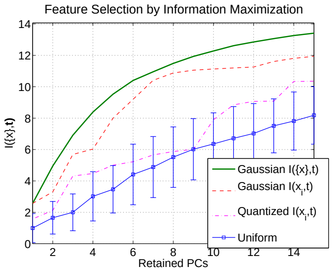
Kulkarni *et al.* [15] introduce the VISTA system, an interactive compilation system which concentrates on reducing compilation time. This system uses a variety of techniques to reduce the number of different compilation sequences evaluated. Their system stores a representation of each program compiled then detects when identical or equivalent code has been generated and only executes code that has not been previously generated. They also prohibit specific optimizations and optimization sequences from being performed if it is unlikely that these optimizations will not change the code. Clearly these techniques are only effective when programs are extremely small, such as those used in embedded domains.

Kulkarni *et al.* [16] also introduce techniques to allow exhaustive enumeration of all distinct function instances that would be produced from the different phase-orderings of 15 optimizations. This exhaustive enumeration allowed them to construct probabilities of enabling/disabling interactions between the different optimization passes. Using these probabilities, they constructed a *probabilistic batch compiler* that determined which optimization should be applied next depending on which one had the highest probability of being enabled. This method however does not consider the benefits each optimization can potentially provide. In contrast, we train our model to based on the impact of optimizations applied, and therefore our technique learns which optimizations are beneficial to apply to “unseen” programs with *similar* characteristics.

Another system to speedup iterative compilation was recently introduced by Cooper *et al.* called ACME [6]. ACME utilizes a technique called estimated virtual execution (EVE) which estimates changes to the execution counts of basic blocks when an optimization that changes the CFG is performed. This is done by inserting a pass into the optimization sequence after each invocation of a CFG-changing optimization and fixes the basic block counts based on the changes. They can then simply model the benefits and disadvantages of applying optimizations by simply multiplying the number of instructions in a block by its dynamic frequency then summing over all blocks. This technique can estimate the performance of very simple models, however this method is vastly inaccurate when estimating the performance of today’s complex machines, especially out-of-order issue processors.

Triantafyllis *et al.* [28] develop an alternative approach to reduce the total number of evaluations on a new program. Here the space of compiler options is examined off-line on a per function basis and the best performing ones classified into a small tree of compiler options. When compiling a new program, the tree is searched by compiling and executing the best path in the tree. As long as the best sequences can be categorized into a small tree, this proves to be a highly effective technique.

Pan *et al.* [21] develop an algorithm called *combined elimination* which selectively turns off optimizations until the best performance is found for a new application. This algorithm was compared to other algorithms for tuning compiler settings [13, 28] and was shown to achieve the same or better performance as these algorithms while dramatically reducing the tuning time. In a recent paper [22], they par-



Most Informative Performance Counters		
1) L1_TCA	2) L1_DCH	3) TLB_DM
4) BR_INS	5) RES_STL	6) TOT_CYC
7) L2_ICH	8) VEC_INS	9) L2_DCH
10) L2_TCA	11) L1_DCA	12) HW_INT
13) L2_TCH	14) L1_TCH	15) BR_MS

Figure 13. Analysis of the importance of the performance counters. The data contains one good optimization sequence per benchmark. *Left:* **Approximate amount of information retained in the selected features. Top curve: our method; lower curves: simpler heuristics. Conventional approach [27] which uses binning and ignores input interactions (dash-dotted curve) is comparable with the uniform random selection.** *Right:* **15 most informative performance counters sorted in the order of the descending importance.**

titioned a program into *tuning sections* and then use combined elimination to find the best combination of optimizations for each of these tuning section. They are able to reduce the time to find good optimization settings from hours to minutes. However, the techniques of partitioning a program and rating methods are orthogonal to the particular search algorithm used and we concluded we could as well use our models to find good sequences even faster. Also, the techniques in this paper could be beneficial during the training data generation stage of our technique. Specifically, the technique to test different optimization settings on a tuning section during a single run of the program would allow us to increase the number of optimization settings we evaluate.

Yotov *et al.* [31] describe a model-based approach to optimize BLAS libraries that can as effective as empirical evaluation. In the later paper [32] they refine analytical models based on the results of the empirical search for the ATLAS library. A local neighborhood search around the best found points is used to further improve the solutions and perform comparable to the ATLAS global search strategy. However, the analytical models require manual tuning and are complicated.

Cooper *et al.* [7] use genetic algorithms to solve the compilation phase ordering problem. They were concerned with finding “good” compiler optimization sequences that reduced code size. Unfortunately, their technique is application-specific. That is, a genetic algorithm has to *retrain* for each program to decide the best optimization sequence for that program. The genetic algorithm builds up chromosomes pertaining to different sequences of optimizations and adapts these for each individual program. Mutations can involve adding new optimizations into the sequence or removing existing ones from the sequence. Their technique was successful at reducing code size by as much as 40%.

Several researchers have also looked at using machine learning to construct heuristics that control a single optimization. Stephenson *et al.* [26] used genetic programming (GP) to tune heuristic priority functions for three compiler optimizations: hyperblock selection, register allocation, and data prefetching within the Trimaran’s IMPACT compiler. For two optimizations, hyperblock selection and data prefetching, they achieved significant improvements. However, a closer look at the results indicate that all the improvement was obtained simply by producing 400 heuristics at random (for the initial population) and choosing the best one. For those cases, there were no further gains obtained from using GP to evolve the generated heuristics. This indicates that these two pre-existing heuristics were not well tuned. The authors also admit that turning off data prefetching completely was preferable and reduced much of their gains. For the third optimization, register allocation, they were able to achieve on average only a 2% increase over the manually tuned heuristic. The results in this paper highlight the diminishing results obtained when only controlling a single optimization.

Cavazos *et al.* [5] describe an idea of using supervised learning to control whether or not to apply instruction scheduling. They induced heuristics that used features of a basic block to predict whether scheduling would benefit that block or not. Using the induced heuristic, they were able to reduce scheduling effort by as much as 75% while still retaining about 92% effectiveness of scheduling all blocks.

Monsifrot *et al.* [19] use a classifier based on decision tree learning to determine which loops to unroll. They looked at the performance of compiling Fortran programs from the SPEC benchmark suite using g77 for two different architectures, an UltraSPARC and an IA64. They showed an improvement over the hand-tuned heuristic of 3% and 2.7% over g77’s unrolling strategy on the IA64 and UltraSPARC, respectively. These results again highlight the need to control the application of multiple compiler optimizations.

7 Conclusions

In this paper we address the problem of predicting good compiler optimizations by using performance counters to automatically generate compiler heuristics. We do this by using machine learning techniques that predict good code transformations to apply given a program's performance counter features. Our technique automates the tuning process and eliminates the need for manual experimentation. Additionally, the heuristics induced by these techniques can generalize to programs that have not been seen before. Using performance counters allows us to apply transformations that will benefit the program being compiled while avoiding optimizations that will degrade performance. Using our models, we can achieve a 10% average speedup over the highest optimization setting in the PathScale compiler on SPEC benchmarks on a recent AMD Athlon much fater than the current state-of-the-art search techniques.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. OBoyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *CGO*, pages 295–305, 2006.

[2] AMD. Compiler usage guidelines for 64-bit operating systems on amd64 platforms. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/32035.pdf, 2006.

[3] R. Azimi, M. Stumm, and R. W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 101–110, New York, NY, USA, 2005. ACM Press.

[4] C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, Oxford, UK, 1996.

[5] J. Cavazos and J. E. B. Moss. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, pages 183–194, Washington, D.C., June 2004. ACM Press.

[6] K. D. Cooper, A. Grosul, T. J. Harvey, S. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Acme: adaptive compilation made efficient. In *LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 69–77, New York, NY, USA, 2005. ACM Press.

[7] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–9, Atlanta, Georgia, July 1999. ACM Press.

[8] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *Journal of Supercomputing*, 23(1):7–22, August 2002.

[9] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. Profileme : Hardware support for instruction-level profiling on out-of-order processors. In *International Symposium on Microarchitecture*, pages 292–302, 1997.

[10] B. Franke, M. O'Boyle, J. Thomson, and G. Fursin. Probabilistic source-level optimisation of embedded programs. In *LCTES'05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 78–86, New York, NY, USA, 2005. ACM Press.

[11] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[13] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 123–132, Washington, DC, USA, 2005. IEEE Computer Society.

[14] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. 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*, page 237, Washington, DC, USA, 2000. IEEE Computer Society.

[15] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 171–182, New York, NY, USA, 2004. ACM Press.

[16] P. A. Kulkarni, D. B. Whalley, G. S. Tyson, and J. W. Davidson. Exhaustive optimization phase order space exploration. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 306–318, Washington, DC, USA, 2006. IEEE Computer Society.

[17] P. S. Ltd. http://www.polyhedron.com, 2006.

[18] R. J. McEliece. *The Theory of Information and Coding*. Addison-Wesley, 1977.

[19] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *AIMSA '02: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, pages 41–50. Springer-Verlag, 2002.

[20] P. Mucci. Papi – the performance application programming interface. http://icl.cs.utk.edu/papi/index.html, 2000.

[21] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO*, pages 319–332, 2006.

[22] Z. Pan and R. Eigenmann. Fast automatic procedure-level performance tuning. In *IEEE PACT*, Seattle, WA, September 2006. IEEE Computer Society.

[23] D. Parello, O. Temam, A. Cohen, and J.-M. Verdun. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 15, Washington, DC, USA, 2004. IEEE Computer Society.

[24] I. PathScale. http://www.pathscale.com, 2006.

[25] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005. special issue on ”Program Generation, Optimization, and Platform Adaptation”.

[26] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 77–90, San Diego, Ca, June 2003. ACM Press.

[27] M. Stephenson and S. P. Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, pages 123–134, 2005.

[28] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 204–215, Washington, DC, USA, 2003. IEEE Computer Society.

[29] R. Vuduc, J. W. Demmel, and J. A. Bilmes. Statistical models for empirical search-based performance tuning. *Int. J. High Perform. Comput. Appl.*, 18(1):65–94, 2004.

[30] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[31] K. Yotov, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 63–76, San Diego, Ca, June 2003. ACM Press.

[32] K. Yotov, K. Pingali, and P. Stodghill. Think globally, search locally. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 141–150, New York, NY, USA, 2005. ACM Press.

[33] 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.