Blind Optimization for Exploiting Hardware Features

Dan Knights, Todd Mytkowicz, Peter F. Sweeney,
Michael C. Mozer and Amer Diwan⋆

Department of Computer Science
University of Colorado, Boulder

Abstract. Software systems typically exploit only a small fraction of the realizable performance from the underlying microprocessors. While there has been much work on hardware-aware optimizations, two factors limit their benefit. First, microprocessors are so complex that it is unlikely that even an aggressively optimizing compiler will be able to satisfy all the constraints necessary to obtain the best performance. Thus, most optimizations use a simplified model of the hardware (e.g., they may be cache-aware but they may ignore other hardware structures, such as TLBs, etc.). Second, hardware manufacturers do not reveal all details of their microprocessors so even if the authors of optimizations wanted to simultaneously optimize for all components of the hardware, they may be unable to do so because they are working with limited knowledge. This paper presents and evaluates our blind optimization approach which provides a way to get around these issues.

Blind optimization uses the insight that we can generate many variants of an application by altering semantic preserving parameters of an application; for example our variants can cover the space of code and data layout by shifting the positions of code and data in memory. Our optimization strategy attempts to find a variant that performs well with respect to an optimization objective. We show that even our first implementation of blind optimization speeds up a number of programs from the SPECint 2006 benchmark suite.

1 Introduction

Computer systems rarely exploit the underlying hardware to its fullest potential. For example, even though many microprocessors can execute 4 or more instructions per cycle per core, it is rare for applications to execute more than 1 instruction per cycle even for brief periods [10] of time. Thus, there is an enormous potential for improving performance: in theory, at least, we should be able

to obtain multi-fold speedup for many applications without counting on any advances from hardware. Unfortunately, this potential is not easy to realize: modern microprocessors are incredibly complex and worse, hardware manufacturers do not reveal full details of their hardware. As a consequence even if compiler writers were extremely knowledgeable about microprocessors in general, they would not be able to fully exploit any particular microprocessor because they do not know all the details of that microprocessor.

For example, code layout affects how the code ends up in the many different hardware structures inside a microprocessor. These hardware structures include instruction queues, L1 instruction cache, L2 cache, instruction TLB, buffers for issuing prefetches, buffers for predicting branches, etc. Given this plethora of hardware structures, even if we knew exactly how they all worked (which we usually do not), it would require tremendous effort to implement an optimization that lays out the program code so that it interacts well with all of them. To address such situations, this paper proposes and evaluates *blind optimization*, a new model for compiler optimizations.

The key insight behind blind optimization is that an optimization can be ignorant of—or "blind" to—the details of the hardware architecture and yet still offer significant performance improvements. Understanding why the performance has improved is not essential, as long as the improvement is significant and reproducible. In contrast, existing compiler optimizations are "knowledge-based" because they exploit domain knowledge of the underlying machine.

To specify an instance of blind optimization, we specify three elements: an *optimization objective*, the space of *program variants*, and an *optimization strategy*. The optimization objective is the metric that we wish to optimize (e.g., run time of the program). The space of program variants is an n-dimensional space in which each point is a variant of the program being optimized. We pick the dimensions so that they only affect the optimization objective and not the program's correctness. The optimization strategy explores the variant space in an attempt to identify a variant that has the best optimization objective. Thus, with blind optimization we can find a variant that performs well without knowing why it performs well. This is why these optimizations are "blind".

This paper makes two main contributions. First, this paper introduces the concept of blind optimization and discusses how one can implement them. Second, this paper demonstrates that one blind optimization, improving code and global data layout, improves the performance of several programs from the SPECint 2006 suite with a maximum speedup of over 12% and an average speedup of 1.58%.

The remainder of this paper is organized as follows. Section 2 motivates our approach. Section 3 describes our approach at a high level and Section 4 describes a particular concrete instance of blind optimization. Section 5 describes our experimental methodology. Section 6 presents results from the concrete instance. Section 7 discusses the implications of our results. Section 8 reviews related work and Section 9 concludes.

2   Motivation

Predicting the performance of a program run is nearly hopeless because it requires us to correctly answer numerous questions. Should we assume that a load will hit in the L1 cache, L2 cache, or L3 cache? Should we assume that an instruction reference will hit in the L1 cache, L2 cache, or L3 cache? Should we assume that the load or next instruction's address is going to hit in the TLB and if not which level of the hierarchical page table will it hit on? Will the branch predictor correctly predict a particular branch? Will we even need to access the branch predictor for a particular branch or will the loop-stream detector avoid that access? These and many other factors determine the overall performance of a program. Given that hardware manufacturers do not reveal all information about their microprocessors some of these questions may be unanswerable.

The difficulty of accurately predicting performance does not bode well for compiler optimizations. To effectively optimize a program, an optimization must predict, using *predictive heuristics*, how code will interact with hardware structures. Because predicting performance is so hard, most predictive heuristics are simple (e.g., they consider the L1 caches but ignore other aspects of the memory hierarchy) and attempt to be a best-guess; others, e.g., Triantafyllis *et al.* [24], have written about the difficulty of coming up with reasonable heuristics. Perhaps, for this reason, it is not surprising that most compiler optimizations offer only modest benefit [13].

For the above reasons, this paper proposes and evaluates blind optimizations, a new technique that does not rely on predictive heuristics.

3   Approach

To specify an instance of blind optimization, we need to specify three elements: the space of program variants, an optimization objective, and an optimization strategy (Figure 1). Intuitively, our approach uses the insight that a program has variants that are behaviorally equivalent but differ in their performance with respect to the optimization objective, e.g., execution time. The optimization strategy navigates this space in an attempt to identify the best variant. This section describes our approach abstractly and Section 4 gives a concrete example of our approach.

3.1   Space of program variants

A *program variant*, P', is a variant of the program being optimized, P, such that P and P' differ only in performance (if at all). Specifically, P and P' always produce the same answer. By specifying a set of dimensions along which the program can vary, we can define a multidimensional space of program variants—hereafter, *variant space*. Each variant corresponds to a point in this space and thus we can represent it by a discrete-valued vector. Figure 2A shows a two-dimensional variant space. Each point in the grid represents a variant.
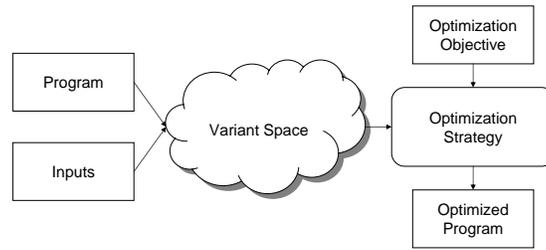
Fig. 1. The blind optimization approach.

The nature of the optimization that we wish to perform determines the dimensions for the variant space. Specifically, we want the dimensions that are actually relevant to our optimization. For example, if we wish to optimize code layout then there may be one dimension for the address (absolute or relative) of each function, loop, or basic block. The dimensions are obviously relevant: changing the address of code blocks clearly affects code layout. To generate a variant (and thus a point in the variant space) we transform the original program or another variant.
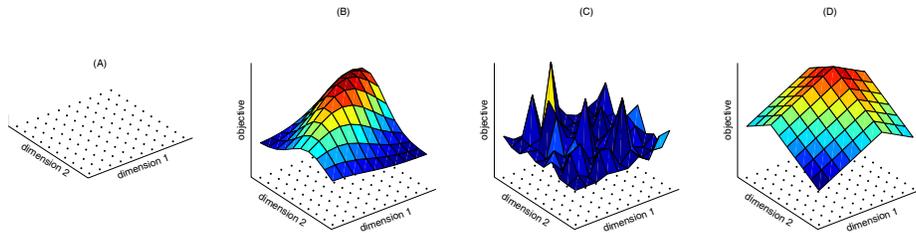

Fig. 2. (A) optimization search space. (B)-(D) potential optimization objective functions.

3.2 Optimization objective

The optimization objective is the metric that we wish to optimize. The obvious objective is to optimize program run time, but one could use other objectives such as program size, number of cache misses or branch prediction accuracy. If we expect that a program's performance will vary significantly with program input, rather than using data from a single execution for each variant, we should use a *mean* execution time from many different inputs.

Figure 2B adds a third dimension, the optimization objective, to the plot in Figure 2A. Specifically, the *objective* value for a point $(d1, d2)$ gives the value of the optimization objective when dimension 1 is $d1$ and dimension 2 is $d2$.

3.3 Optimization strategy

The optimization strategy navigates the variant space in an attempt to identify the most efficient variant. We can exploit techniques from the numerical optimization, machine learning, and operations research literatures to identify possible optimization strategies. However, unlike many optimization problems in those domains, the space we are optimizing over is intrinsically discrete, and therefore we cannot use continuous optimization techniques.

If the variant space is small then we can use exhaustive variant generation: i.e., try all the variants and pick the best one. However, variant spaces in our domain are rarely small enough to allow an exhaustive approach. If the optimization objective has structure then we can use smarter approaches (described below). If it has no structure (e.g., Figure 2C) then the best we can do is to use random search; i.e., pick variants at random and use the best one. On the other hand, if the optimization surface is relatively smooth (e.g., Figure 2B), we can use hill climbing approaches such as genetic algorithms. Unfortunately, our prior work has shown that the optimization surface is rarely smooth: a small change in one dimension can significantly change the optimization objective [18].

From the techniques described above, only the random approach seems feasible. Fortunately, in some cases we can actually do better. For example, if we have reason to believe that the dimensions contribute independently to the optimization objective (e.g., Figure 2D) we can explore one dimension at a time and then combine the results to obtain a variant that performs well. Specifically, this situation corresponds to the case where the optimization objective is a linear combination of functions of the individual dimensions, i.e., $o(\mathbf{x}) = \sum_i f_i(x_i)$, where $o(.)$ is the objective function, $\mathbf{x}$ is the vector corresponding to a variant, and the f_i are a set of functions specifying the relationship between the variant's value on dimension i and the optimization objective. Exploiting this relationship turns a $\mathcal{O}(D^V)$ search into an $\mathcal{O}(DV)$ search, where D is the number of dimensions and V is the number of distinct points along each dimension.

As discussed later (Section 4), the assumption behind the above approach hold for at least some blind optimization scenarios. However, even if the assumption behind the above approach does not hold (we show that sometimes it does not), we may have sufficient domain knowledge to express $o(\mathbf{x})$ as a function of lower order terms, e.g., involving pairs of dimensions. In this paper, we explore blind optimization and thus we do not inject any domain knowledge into our approach. It may eventually turn out that some domain knowledge is beneficial.

4 Implementation

Memory system performance is well known to be one of the main bottlenecks for program performance. Thus, our first use of blind optimization is to improve

the memory behavior of programs. Specifically, our optimization aligns code and global data to improve program performance. Such alignment can affect how the code and data interact with many different hardware structures. For example, if a cache block is 64 bytes and a hot loop is less than 64 bytes, then the loop fits in a cache block if it is aligned correctly; however, if the loop starts in the middle of the cache block then it may spill over to the next cache block which may be detrimental to performance. Because there are many different hardware structures that may be affected by this alignment, it will be difficult to analytically determine the ideal alignment for code and data. Thus, this optimization is a perfect candidate for the blind approach.

As described in Section 3, blind optimization requires us to specify the following components: a variant space, an optimization objective, and an optimization strategy. We now describe these components in detail.

4.1 Variant Space

Our variants differ in the alignment of code and global data to a 64-byte boundary. Our implementation generates variants by shifting functions and global variables. To keep the variant space manageable, we changed the alignment of only hot functions (functions that account for 95% of the total execution time in our training run) and up to 10 randomly picked global variables.

Even within the limited scope of code and global variable alignment, there are many alternatives that we could have pursued. For example, we could have aligned to a 4K boundary instead of a 64 byte boundary to get better alignment to page-level structures. Also, we could have moved code at different granularities (e.g., basic blocks or loops). We will explore these variants in future work.

The variant space has one dimension for each function and global variable, and the values along that dimension are the integers between 0 and 63 (i.e., we use the address of the function or variable modulo 64). We represent a particular variant in this space using a D-dimensional integer vector, where D is the number of functions and global variables that we used.

To generate a particular variant, we first compile the program using gcc (with optimization level -O3) to generate a single assembly file for the entire program (using -combine -S)[1]. Then, we insert `.p2align` and `.byte` directives in the assembly file to affect an alignment. For example, if we want the alignment of function `G` to be 1 byte off from a 64 byte boundary, we insert `.p2align 8` and `.byte 1` before the start of the function. The `.p2align` forces alignment to a 64 byte boundary and the `.byte` directive shifts the following code by 1 byte; thus, `G`'s address modulo 64 will be 1. Finally, use gcc to generate the executable from the instrumented assembly file. We use a similar technique to adjust the alignment of global variables. Using this technique, we can independently control the alignment of each function and global variable.

---

[1] In order to get the entire suite of SPEC C INT 2006 programs to compile with the -combine gcc mode we had to alter a few function headers for most of the programs. We did not change any logic of the code.

4.2 Optimization Objective

We used the program run time as the optimization objective. We measured the run time using hardware-performance monitors and used multiple runs to obtain statistically significant results (Section 5).

4.3 Optimization Strategy

As we discussed in Section 3 we can either use an exhaustive approach or an approach that relies on some structure in the variant space (e.g., linearity). We first show that at least some programs exhibit structure that we can exploit and then describe the two approaches.

Do dimensions independently affect run time? To see if the variant space has structure that we can exploit, we tested if the assumption in Section 3.3 holds: i.e., do the dimensions contribute independently to the run time (e.g., Figure 2D) or do they interact with one another and their interactions affect the run time (e.g., Figure 2B,C). Independence allows for efficient optimization strategies (linear in the number of dimensions) whereas interactions may require exponential search. We show that we can assume independence at least for some of our benchmark programs.

To test for independence, we produced and evaluated a large number, R, of random points in the variant space. For each run, r, we obtained a run time, t_r and a vector $\mathbf{v}_r = \{v_{r,1}, v_{r,2}, ..., v_{r,D}\}$ where $v_{r,i}$ gives the value of the ith dimension in run r.

Next, we classified variant run times. Specifically, as we change the alignment of a function or global variable we do not see a smooth change in the run time; instead, we may see only a few different run times (usually 2 to 5) and many different alignments can produce the same run time. For example, the odd alignments of a function may all yield a "fast" run and the "even" alignments all yield a "slow" run, with nothing in between. This classification induces a clustering on the alignment of each function or global variable; in the above example, the odd alignments will be in one cluster and even alignments in another cluster. Our clusters were often surprising: for example some clusters included a mixture of odd and even alignments (e.g. function `foo` aligned to a 13 byte boundary); we would probably not have guessed these clusters using a knowledge-based approach.

We then used the clustering to convert the variant vectors into vectors of indicator variables, $\mathbf{q}_r = \{q_{r,1}, q_{r,2}, ..., q_{r,D}\}$, where $q_{r,i}$ is 1 if $v_{r,i}$ was in cluster 1 and 0 otherwise. The linear model we wish to produce is now:

$$\hat{t}_r = \sum_d w_d \, q_{r,d}$$

where $\mathbf{w}$ is a vector of weights (coefficients) and d spans D, the number of dimensions in the vector space. If $\hat{t}_r$ predicts the actual t_r accurately for a large

number of runs $R >> D$, it implies that function and global variable alignments contribute independently to the total run time. The constraint $R >> D$ is necessary to ensure that the model, which has D free parameters, is simple relative to the number of data points, R, that it explains.

We develop the simplest linear model via a greedy add-one-in regression. In other words, at each step, we add the dimension that yields the best improvement in the squared error between $\hat{t}_r$ and t_r. We stop when adding another dimension does not yield a significant improvement in squared error.
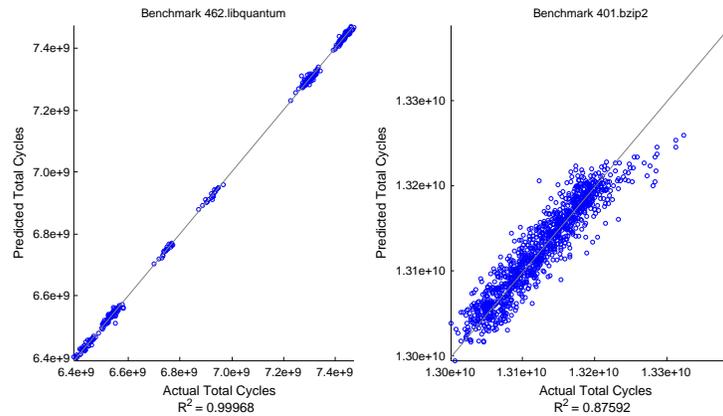

Fig. 3. Predicted versus actual run times.

Figure 3 compares actual run time, t_r, to the model's prediction, $\hat{t}_r$. Each point on the scatter-plot corresponds to a single run, r. We see that we get a good linear fit for libquantum while the fit is much worse for bzip2. Thus, while some of our programs are amenable to a linear model, others are not. For this reason, we explore two approaches in our experiments: (i) *random search* assumes that the optimization objective does not exhibit a structure that we can exploit; and (ii) *independent dimension search* assumes that the dimensions independently contribute to the optimization objective.

Approach for random search. Random search simply tries many variants and chooses fastest variant as the optimized program. Random search is not as naive as it might sound. Ordinarily, one would not expect a random search in a space of 64^D variants to turn up anything close to the optimal variant. However, our earlier clustering results suggest that the real space is actually much smaller than 64^D and thus random search with even a modest number of variants may actually produce good results.

Approach for independent dimension search. Unlike random search, independent dimension search does not simply pick the best variant out of the ones that it has tried; instead it synthesizes a (possibly as yet untried) variant by analyzing the variants it has seen. It works as follows:

1. For a program whose variant space has D dimensions, and each dimension has 64 possible alignments, randomly select a set of $64D$ variants in the variant space subject to the constraint that over the set, all 64 alignments for each dimension occur with equal frequency.
2. Measure the run time of each variant.
3. For each dimension, d, compute the mean run time for each possible alignment. This involves computing the average run time over all random variants whose value for dimension d is a, for $d = 1, ...D$ and $a = 0, 1, 2, ..., 63$. Let $\bar{t}_{d,a}$ denote the mean run time for dimension d aligned to byte a.
4. For each dimension d, choose the best alignment $a_d^* = \arg\min_a \bar{t}_{d,a}$.
5. Form a new variant in which each dimension's value is a_d^*. This variant will be the optimized program under the assumption of independence.

In future work, we will likely opt for hill climbing search which is based on a small number of equivalence classes instead of 64 possible alignments.

5 Methodology

With all aspects of our measurements, we followed best practices so as to avoid perturbing our data. Specifically, we conducted all our experiments on minimally-loaded machines and used only local disks. We ran each benchmark N times—where N is such that the 95% confidence interval of the mean is 0.5% of the mean itself. N was 3 for most of our benchmarks. We used PAPI [3] (version 3.5.0) to capture the cycle counter before and after a benchmark runs. We used the default (as per SPEC) linking order for all benchmarks. We used *gcc* version 4.2.1 and optimization level *O3* to compile our benchmarks. Finally, we ran our programs in an empty environment (`env -i`) and turned off the kernel's address randomization.

Table 1 presents SPECint 2006 [22] benchmarks that we use (we omitted benchmarks not written in C). For each benchmark, Table 1 gives the number of variants that we generated and the number of inputs that we used. We used the *ref* and *train* inputs provided by SPEC.

Because of the large running times of the SPEC programs (total machine time was over 525 hours) and the large number of program variants required by blind optimization we used three similar Intel Core 2 workstations. Each workstation runs the Linux operating system on a Core 2 processor. We ran all experiments for a particular benchmark on the same machine to remove the possibility of introducing a confounding variable into our analysis.

6 Results

In this section, we evaluate our first instantiation of blind optimization.

Benchmark	Description	# Inputs	# Variants
bzip2	Compression algorithm	10	100
gcc	C Compiler	10	100
gobmk	Go game	7	100
hmmer	Computational biology DNA search	4	100
h264ref	Video encoding	10	100
lbm	3D Fluid dynamics	3	100
libquantum	Quantum computer simulator	10	100
mcf	Single-depot vehicle scheduler	3	100
milc	4D Lattice simulations	3	100
perlbench	Scripting language interpreter	5	100
sjeng	Chess program	3	100

Table 1. Benchmark programs.

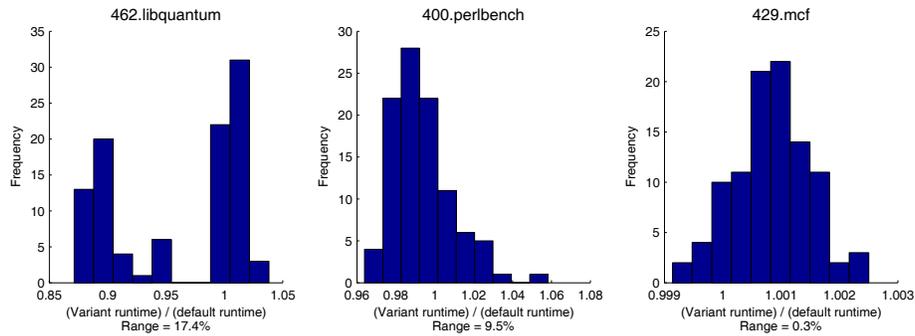## 6.1 Are programs amenable to code- and global data-layout optimization?

Fig. 4. Distribution of run-times.

Figure 4 shows that improving code- and global data-layout can significantly affect program performance. The height of a bar gives the number of variants that have a particular execution time (indicated by the x-axis label). We normalize all execution times to the execution time of the default variant. We present the histograms only for three benchmarks due to space limitations: libquantum and perlbench with a wide range of 17.4% and 9.5% respectively, and mcf with its narrow range of 0.3%. From these histograms we conclude that depending on which variant *gcc -O3* actually generates, our approach may be able to speed up these programs by up to 17.4%. On the other hand, our optimization will not help some benchmarks, such as *mcf*.

In general, we have found that *gcc -O3* does only slightly better than a randomly chosen program variant. When averaged across all inputs for all benchmarks, we found that *gcc -O3* was slower than the average variant for five out of the eleven benchmarks. This is remarkable, since it suggests that *gcc*'s domain knowledge is not helpful; thus blind optimization is a promising alternative.

Benchmark	random: potential % speedup	random: observed % speedup	indep.: observed % speedup
bzip2	1.04	0.93	0.81
gcc	0.23	0.20	-2.19
gobmk	0.49	0.47	-0.48
hmmer	2.72	0.32	0.76
h264ref	0.12	0.05	0.04
lbm	0.70	-0.14	0.17
libquantum	12.61	12.61	12.46
mcf	0.51	-0.26	0.02
milc	2.24	1.93	1.43
perlbench	1.17	0.24	-0.29
sjeng	1.10	1.10	0.53

Table 2. Cross-validation results for random and independent models on all benchmarks

Table 2 presents the benefit due to blind optimization of code- and global data-layout for each benchmark. The "random: % observed improvement" and "indep: % observed improvement" give percertage speedups (over the default variant) using random search and indpendent dimensions search respectively. We obtained these speedups using n-fold cross-validation. For each of the n inputs of a given benchmark (shown in the "# inputs" column in Table 1), we used the remaining $n-1$ inputs to choose (random search) or produce (independent dimensions search) the best program variant, and then measured the speedup obtained on the nth hold-out input. We present the average speedup over all n folds. This methodology, commonly used in the statistics literature, ensures that we do not use the same inputs for training as for evaluation.

From Table 2 we see that random search speeds up 9 of the 11 benchmarks and slightly slows down two benchmarks (lbm and mcf). Moreover, three programs show significant (more than 1%) speedups: libquantum, milc, and sjeng. These speedups are significant because they come on top of code that *gcc* has already optimized.

From the data for independent dimensions search (Column "indep: observed % speedup") we see that it outperforms random search for only three benchmarks (hmmer, lbm, and mcf). This is not surprising; as Figure 3 shows the independent dimensions assumption does not always hold.

6.2 Is the fastest variant on one input the fastest variant on another?

So far, all our results use cross-validation; i.e., we evaluate and train on different inputs. The "random: % potential improvement" column shows the speedup we would get if we trained and evaluated on the same input. In other words, it gives the upper-bound for how well random search can do. Comparing the "random: % potential improvement" and "random: % observed improvement" columns tells us the extent to which the optimization generalizes across inputs. We see that for many benchmarks it does but for some benchmarks (particularly hmmer, lbm, mcf, and perlbench) it does not. In other words, the inputs for these benchmarks behave differently enough from each other that we cannot fully translate results from one input to another input. This underlies the need to have a good set of training inputs for blind (or any profile-guided) optimization.

6.3 Do our results generalize across machines?

To answer this question, we used random search to find the best variant on one (training) machine and then compared that variant to the default variant on another (test) machine. Both machines use the Core2 chip, but with different amounts of memory and different clock speeds. We found that the speedups on one machine were highly correlated with speedups on the other machine, with an R^2 value of 0.81. The linear model for the libquantum, for example, achieved a 12.61% improvement over *gcc* on the training machine and a 12.51% improvement on the test machine. Thus, at least in some cases, our results generalize across machines.

6.4 Do our results generalize across compilers?

To see if the benefit due to blind optimization was an artifact of something in *gcc*, we repeated the experiments for libquantum (the benchmark with the greatest speedup) using Intel's *icc* compiler. Blind optimization was able to speed up libquantum by 4.63%; while this speedup is smaller than what we observed with *gcc* it is still significant. Thus, blind optimization is useful even for code compiled using *icc*.

7 Discussion

The performance of a program depends not just on characteristics of the program but also on characteristics of the underlying system. Thus, we do not view blind optimizations as something that software manufacturers do just before they ship out their code; instead it is something that occurs at installation time. Indeed, it may be worthwhile to treat blind optimizations analogously to "disk defragmentation": periodically, when the machine is idle, we can rerun blind optimizations on the most performance critical applications. Because blind optimizations do not need the source code, this approach is feasible; moreover, as clients of the software use the system, we can record client inputs and use those inputs to explore the variant space. In this way, the re-optimization will be customized to how clients actually use the software.

8 Related Work

Compiler optimizations have obviously been an active area of research for several decades. Broadly speaking, prior work falls in four categories: optimization-space exploration, machine learning to derive predictive heuristics, search-based optimizations, and knowledge-based optimizations.

8.1   Optimization space exploration

This area solves the following problem: given the following set of optimizations, which ones should we use and in what order should we apply them? For the most part, once they have picked the set and order of optimizations, that order is used unchanged for all programs.

Pan *et al.* [19] use an offline search to find an optimization combination that works well for a training set; this combination is used to optimize subsequent programs. Triantafyllis *et al.* [24] uses trials at compiler-construction time that produces a hopefully small set of configurations that perform well for a set of training benchmarks. When compiling a new program, they pick one of the configurations from this set; this set is organized hierarchically which helps to quickly identify the best one for the current program.

Given a set of optimizations and underlying system (microprocessor, OS, etc.), work in this area is invaluable for picking combinations that work well together on that system. Blind optimization compliments this area by refining the optimized binaries at a fine-grained level (i.e., applications of individual transformations).

8.2   Machine learning to derive predictive heuristics

This area uses machine learning to learn predictive heuristics which the compiler uses when optimizing programs.

Cavazos and Moss [5] use supervised learning to learn heuristics for whether or not a basic block is worth scheduling. This heuristic helps focus scheduling effort on blocks that may actually benefit from it. This approach depends on a simulator that can evaluate different schedules; thus the simulator is the "supervisor". Cavazos and O'Boyle [6] use genetic algorithms to find the setting of inlining parameters; they use these settings for subsequent compilations. Singer *et al.* [21] build decision trees to decide which garbage collection to use for an as-yet unseen application. To pick the garbage collector for an unseen application, they identify training applications that were similar to this application and use the garbage collector that performed best for the training application.

These techniques free the compiler writer from having to come up with heuristics. However, they assume that program-independent features are enough to base predictive heuristics on. In contrast, blind optimization does not depend on predictive heuristics.

8.3   Search-based optimizations

Search-based optimizations attempt to obtain good performance by exploring a space of optimizations and picking the best point in the space for a given piece of code. Blind-optimization is a search-based optimization technique.

Massalin's superoptimizer [14], for example, exhaustively explores instruction combinations to find the shortest sequence that behaves the same as the sequence being optimized. McGovern *et al.* [17] try many different schedules of

one basic block at a time and pick the one that gives the best performance. Because McGovern *et al.*'s technique works on one basic block at a time, it requires a simulator to estimate the performance of the basic block. Cooper *et al.* [7] use biased random sampling to try many different compilation sequences (i.e., different orders for optimizations) to identify an order that gives the best performance for the program being compiled. Lau *et al.* [12] effectively implement Cooper *et al.*'s approach in an online setting. Given multiple versions of a method (variants), each optimized differently, Lau's approach uses sampling to pick the best variant. It uses exhaustive search since it collects data on all the variants.

All of the above work can be thought of as examples of the blind optmization approach. For example, the Massalin's superoptimizer uses program size as the optmization objective and the machine's instruction set as the variant space. In contrast, our instantiation of the blind optimization approach either (i) uses a much larger variant space where exhaustive search is simply not possible; or (ii) attempts to identify structure in the variant space which enables us to efficiently search the space.

8.4 Knowledge-based optimizations

Knowledge-based optimizations attempt to improve performance by incorporating significant domain knowledge about what makes code efficient or inefficient on the underlying system. This work falls in two categories: dynamic and static.

Dynamic knowledge-based optimizations improve the performance of code while it is running. Adaptive optimizations [1] track which code is hot and optimize only that code. Feedback-directed optimizations [2] continually reevaluate optimization decisions while the optimized program is running. Both adaptive and feedback-directed optimizations avoid having to predict which code is slow: they know which code is slow since they have measured it recently. However, unlike blind optimization, they still need to predict the benefit of an optimization on the subsequent performance of the code.

Static knowledge-based optimizations requires deep knowledge of the underlying system to optimize code using profiling data. Required knowledge of the underlying system for these approaches to work include: knowledge that the instruction cache is direct-mapped [15, 8], knowledge of the size of the instruction cache [9, 16] knowledge of the branch predictor [11, 20, 23, 4]. In contrast, blind optimization requires no knowledge of the underlying hardware.

9 Conclusions

We have introduced blind optimization, a useful new approach for optimizing programs to better utilize the underlying hardware. We have demonstrated this approach with a single example: improving code and global-data layout. We have shown that even this single example yields statistically significant speedups (average 1.58%) and in one benchmark, large (12%) speedup. These results are

exciting since we are improving code that *gcc* has already optimized (even with respect to its alignment) to the best of its ability.

References

1. Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, October 2000. OOPSLA.
2. Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of java. *SIGPLAN Not.*, 37(11):111–129, 2002.
3. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *SC*, Dallas, Texas, November 2000.
4. B. Calder and D. Grunwald. Reducing branch costs via branch alignment. In *ASPLOS*, Oct. 1994.
5. John Cavazos and J. Eliot B. Moss. Inducing heuristics to decide whether to schedule. In *PLDI*, pages 183–194, New York, NY, USA, 2004. ACM.
6. John Cavazos and Michael F. P. O'Boyle. Automatic tuning of inlining heuristics. In *SC*, page 14, Washington, DC, USA, 2005. IEEE Computer Society.
7. Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, 2002.
8. Nikolas Gloy, Trevor Blackwell, Michael D. Smith, and Brad Calder. Procedure placement using temporal ordering information. In *MICRO*, pages 303–313, 1997.
9. Amir H. Hashemi, David R. Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. In *PLDI*, pages 171–182, 1997.
10. M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *OOPSLA*, 2004.
11. Daniel A. Jimnez. Code placement for improving dynamic branch prediction accuracy. In *PLDI*, pages 107–116. ACM Press, 2005.
12. Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. Online performance auditing: using hot optimizations without getting burned. *SIGPLAN Not.*, 41(6):239–251, 2006.
13. Han Lee, Daniel von Dincklage, Amer Diwan, and J. Eliot B. Moss. Understanding the behavior of compiler optimizations. *Softw. Pract. Exper.*, 36(8):835–844, 2006.
14. Henry Massalin. Superoptimizer: a look at the smallest program. *SIGPLAN Not.*, 22(10):122–126, 1987.
15. Scott Mcfarling. Program optimization for instruction caches. In *ASPLOS*, pages 183–191. ACM, 1989.
16. Scott Mcfarling. Procedure merging with instruction caches. In *PLDI*, pages 71–79, 1991.
17. Amy McGovern, Eliot Moss, and Andrew G. Barto. Building a basic block instruction scheduler with reinforcement learning and rollouts. *Mach. Learn.*, 49(2-3):141–160, 2002.
18. Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *ASPLOS*, 2009.
19. Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO*, pages 319–332, Washington, DC, USA, 2006. IEEE Computer Society.

20. Karl Pettis and Richard C. Hansen. Profile guided code positioning. In *PLDI*, pages 16–27, June 1990.

21. Jeremy Singer, Gavin Brown, Ian Watson, and John Cavazos. Intelligent selection of application-specific garbage collectors. In *ISMM*, pages 91–102, New York, NY, USA, 2007. ACM.

22. Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks. http://www.spec.org/cpu2006/.

23. Hiroyuki Tomiyama and Hiroto Yasuura. Code placement techniques for cache miss rate reduction. *ACM Trans. Des. Autom. Electron. Syst.*, 2(4):410–429, 1997.

24. Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *CGO*, pages 204–215, Washington, DC, USA, 2003. IEEE Computer Society.