McGill University
School of Computer Science
Sable Research Group

Sable
McGill

A Stochastic Approach to Instruction Cache Optimization

Sable Technical Report No. 2008-4

Maxime Chevalier-Boisvert and Clark Verbrugge
{mcheva,clump}@cs.mcgill.ca

December 17, 2008

Contents

1 Introduction **3**

2 Related Work **4**

3 Design **6**

3.1 Optimization Framework 6

3.2 Stochastic Search Algorithm 7

3.3 Cache Coloring Algorithm 8

3.4 Padding Overhead 9

4 Experimental Analysis **9**

4.1 Performance 10

4.2 Cache Misses 12

4.3 Convergence 13

5 Conclusions and Future Work **14**

List of Figures

1 Information flow among the components of our framework 6
2 Memory layout modification in our framework 7
3 Optimized running times relative to original time (Athlon64, O3) 10
4 Optimized running times relative to original time (Athlon64, O0) 11
5 Optimized running times relative to original time (Pentium III, O3) 11
6 Optimized running times relative to original time (Pentium IV, O3) 12
7 Average relative fitness over time 14

List of Tables

1 Description of our benchmarking platforms 9
2 Complexity of our benchmark programs 10
3 Cache and performance effects of the genetic algorithm (Athlon64, O3) 13
4 Cache and performance effects of the coloring algorithm (Athlon64, O3) 13

Abstract

The memory alignment of executable code can have significant yet hard to predict effects on the run-time performance of programs, even in the presence of existing aggressive optimization. We present an investigation of two different approaches to instruction cache optimization—the first is an implementation of a well-known and established compile-time heuristic, and the second is our own stochastic, genetic search algorithm based on active profiling. Both algorithms were tested on 7 real-world programs and speed gains of up to 10% were observed. Our results show that, although more time consuming, our approach yields higher average performance gains than established heuristic solutions. This suggests there is room for improvement, and moreover that stochastic and learning-based optimization approaches like our own may be worth investigating further as a means of better exploiting hardware features.

1 Introduction

Program performance on modern processors is predicated on good use of advanced hardware components, and in particular instruction (and data) caches. Hardware improvements, most notably increasing cache associativity, are the main source of greater efficiency in this regard, but in practice, program performance can still show strong variability due to i-cache behaviour [9], suggesting there is an opportunity for general improvement. Hence, various compiler-based approaches have been devised to rearrange procedures in memory in order to optimize instruction cache usage [14, 11, 8, 3, 12].

We investigate the potential for instruction cache optimization through two different approaches. Following the main heuristic used in i-cache optimization, minimizing *first-order* (immediate caller-callee) cache conflicts, we develop an implementation of a well-known *cache coloring* approach [11]. We complete the extension of this algorithm to handle set-associative caches in order to examine its performance when applied to more modern architectures. We also examine a new stochastic optimization based on a machine learning approach: genetic algorithms. Here, a set of candidate solutions are evolved toward an optimal solution, as measured by program execution time. This design functions as a general optimization heuristic driven by actual performance, and follows current interest in machine learning approaches to optimization [2, 5].

We evaluate both techniques using a common research framework that accommodates multiple languages and architectures. Results are given by measuring performance on multiple architectures for a set of non-trivial benchmarks, and also consider the impact of other optimizations. While there is significant variation in the ability of either algorithm to accommodate individual benchmarks, we show that speed improvements of up to 10% can be had with our genetic algorithm approach. The coloring implementation is less successful, but is still able to offer some useful improvements, suggesting that performance increases are both possible and that they can be significant, even in the context of existing, aggressive hardware and software optimizations.

Specific contributions of this work include:

- We explore the use of genetic algorithms for i-cache optimization. With increased complexity due to evolving hardware, OS, and software designs, learning-based approaches have significant promise as a means of increasing performance. We are able to show a consistent if small improvement (0.5%), and up to 10% for specific benchmarks.
- The evaluation of previous work on i-cache optimization based on cache line *coloring* [11] in the

context of multiple architectures. We extend the design to handle set-associativity, and show that while performance differences are generally variable and within noise, significant positive improvements, up to nearly 6% can be achieved.

- The design and implementation of a research framework suitable for exploring instruction cache optimization algorithms. Our approach is language and hardware agnostic, accommodating any environment where assembly code can be produced and edited.

In the next section we discuss related work with respect to instruction cache optimization and analysis. This is followed by details on our own framework and approach in Section 3, and experimental results using our design in Section 4. Section 5 concludes and describes possible future work.

2 Related Work

The designs we investigate optimize instruction cache performance by choosing an optimal arrangement of procedures (or smaller code chunks) in memory, trying to locate procedures at specific relative cache boundaries in order to ensure that the execution of repeated code sequences is likely to encounter all instructions in the cache, and avoid expensive cache misses. Truly optimal solutions are impractical for large bodies of code, and so most designs are based on heuristics, based on data from run-time profiles. Pettis and Hansen's algorithm, for instance, simplifies the problem by heuristically focusing on minimizing first-order cache conflicts, as found from a profile-driven (weighted) call-graph [14]. Their design includes procedure splitting and basic block reordering, and they show the best benefit is achieved with these additional features. Hashemi et al.'s *coloring* approach operates in a similar fashion, but takes into account cache and procedure parameters, offering a 17% reduction in cache misses over Pettis and Hansen's algorithm [11]. Coloring approaches also more naturally extend to modern, set-associative cache designs, as we investigate here.

More advanced designs make use of calling context information. By profiling method call-chains, the temporal order of method invocation can be considered in the procedure layout, albeit with significantly increased profiling cost and complexity. Gloy et al. consider several designs based on caller-callee relations gathered from temporal profiles in conjunction with the basic heuristic of minimizing first-order conflicts [8]. They accommodate multi-level caches, and are able to show a 13–31% improvement over Pettis and Hansen [14]. (Limited) experiments show, however, that this does not trivially extend to set-associative cache designs. Guillon et al. later improved this algorithm, reducing the incidental code growth with a better memory placement algorithm. This reduces the code expansion by factor of over 20, while maintaining a similar miss reduction [10]. Kalamatianos et al.'s approach combines temporal relations of method calls with coloring. Profile data is used to build a *Conflict Miss Graph,* estimating worst-case cache conflicts between procedures. Once pruned, this again allows for a coloring approach to minimize immediate caller-callee conflicts [13]. Maintaining the profile information in temporal form can also offer advantages; Bartolini and Prete, for example, model the effect on the cache using reduced (sampled) program traces at each stage [3]. This allows dynamic information to be retained at each step, as the procedure layout is incrementally built. As with all cache optimizations this is less effective with large and highly associative caches; Bartolini and Prete, however, show that by reducing cache pressure, programs can operate as efficiently with a cache that is 2–4× smaller, an important property when considering power requirements or embedded system constraints.

Instruction cache optimization has a natural application to JIT-based systems. If code is dynamically generated or loaded, then it can be dynamically arranged in some optimal fashion. Chen and Leupen's *Just In Time Code Layout,* for example, incrementally copies the program into memory, loading procedures as they are first invoked. This technique requires no advanced profiling, and heuristically improves performance by arranging procedures in *activation order* [6]. Chen and Leupen are able to show cache performance similar to Pettis and Hansen, but with a greatly reduced memory footprint. Scales' *dynamic procedure placement* improves upon this by allowing the activation order to be reset to ensure a new group of procedures are contiguous [15]. More recently Huang et al. have examined *dynamic code reordering* in a Java Virtual Machine. They are able to take advantage of the existing profile data in JikesRVM to implement a low overhead, practical design; the main improvement, however, comes code space reorganization more than cache miss minimization, and results on their benchmarks are mixed [12].

A central concern in instruction cache optimization is the way caches vary in design, and individual programs also vary significantly in how well they exercise them. As shown by several analytical and simulation-based approaches in the literature, good modeling and analysis of cache behaviour is useful for guiding optimization. Ghosh et al., for instance, introduce a Cache Miss Equation framework, which models loop nesting with a system of equations, the solutions of which indicate the number of misses [7]. This precise model allows for some optimal procedure layout solutions to be calculated. For more general situations an experiment or simulation is required, and several authors have reported on the extent of variation in cache activity. Bradford and Quong, for example, gave a detailed examination of instruction and data cache miss rates for the SPEC92 benchmarks under various simulated cache parameters [4]. They show there can be extensive variation, depending on cache size, associativity, and code generation flags. Interestingly, at least for their benchmarks, there is limited input sensitivity. Simulation approaches such as this allow for high accuracy, at a cost of increased experimental time. This can be addressed through improved instrumentation and data aggregation [19], or by approximation. The latter is considered by Vera et al., who demonstrate a framework for fast cache behaviour analysis by using samples of loop iterations, with tunable accuracy and overhead tradeoffs [18]. The coloring algorithm we consider has only a simple and coarse model of cache design, while the learning algorithm has an ideal model in the sense that it adapts to the actual performance of the underlying architecture.

Our stochastic design uses techniques from the domain of machine learning. Although relatively new, a variety of machine learning approaches have been applied to several complex program optimization problems. Genetic algorithms, for instance, have been used effectively in loop tiling optimization, offering fast convergence and near optimal solutions, albeit in small and constrained situations [1]. Stephenson et al. apply genetic algorithm techniques in the more general search for optimization heuristics; they examine register allocation and data prefetching in particular, and show promising, but variable results, suggesting sensitivity to genetic algorithm parameters [17]. Other learning approaches have also met varied success. Sequeira et al.'s use of *annealing* to improve spatial locality showed limited and variable improvement [16], while Agakov et al. successfully use investigate multiple learning techniques to greatly speed up the search for optimization hot spots in the context of iterative optimization [2]. Cavazos and O'Boyle's approach to generating method-specific optimization sets in JikesRVM using logistic regression was able to improve performance by 29% to 33% [5]. There is significant promise in using learning techniques in optimization, although effective application clearly depends on many complex factors.

3 Design

The heuristic instruction cache optimization algorithm we have implemented is meant to serve as a point of comparison for our own stochastic optimization algorithm. In this section, we present the optimization framework on which both implementations are based, the motivation behind the two approaches, as well as the implementation details of our code rearrangement system. We also discuss our investigation of code-expansion due to “padding” as a potential source of overhead introduced by our implementation.

3.1 Optimization Framework

We have designed a framework to facilitate the implementation and the comparative testing of multiple instruction cache optimization strategies. This framework is implemented in the Python programming language and incorporates interfaces to tools such as the GNU compilers, the GNU profiler and Valgrind. Code optimizations are implemented as plug-in components that can access existing components of the framework.

Our framework is designed around the transformation of benchmark programs. These programs are provided as input in source code form, and can then be compiled either directly into executable binary form, or into assembly (see fig. 1). Interfaces are provided to gather information about the assembly source (i.e., location and type of symbols in the said source), and to gather profiling data from executable binaries (i.e, running time, run-time call-graph, “hot functions”, cache use profiling).

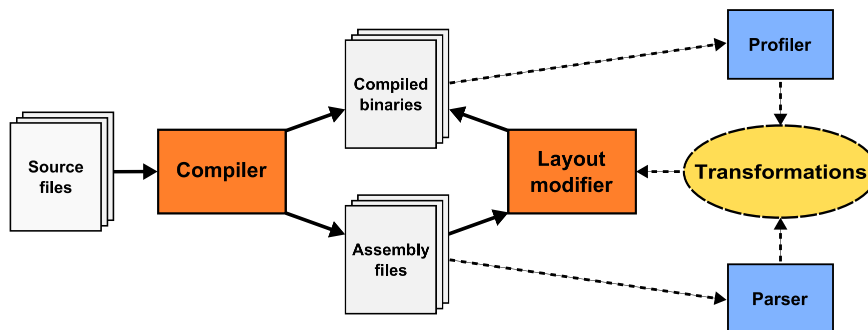

Figure 1: Information flow among the components of our framework

Code optimizations, which we refer to as “transformations”, can use this information in deciding how to transform the assembly source. Because we are specifically interested in instruction cache optimizations, our framework includes a layout modifier component which can be used to modify assembly source so as to change the memory alignment of functions. This component can align functions so that they will map to specific cache lines, for example. Once transformed, assembly source can be compiled into binary form and profiled. It can then be transformed again based on new profiling data.

Figure 2 illustrates the basic idea behind the layout modifier component of our framework. In this example, four functions A, B, C and D are linearly mapped in memory, and take up 1, 1, 3 and 2 cache lines, respectively. Functions C and D originally map to cache lines 0 and 3 of an imaginary 4-line direct-mapped cache. To map functions C and D to cache lines 1 and 2, our layout modifier

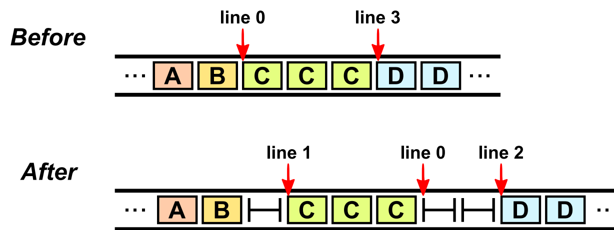

Figure 2: Memory layout modification in our framework

would insert "padding" space equivalent to one cache line in between B and C, and space equivalent to two lines between C and D. The resulting mapping has the same linear order as the original, but takes up slightly more memory. This approach has the advantage that it does not affect the branch prediction behavior of the processor.

Thus far, our framework incorporates support for programs implemented in C, C++ and Fortran, through the GNU `gcc`, `g++` and `gfortran` compilers, respectively. It also incorporates a benchmarking system that can automatically test multiple code optimizations on a sequence of benchmark programs and record the running-time of each program before and after optimization into a spreadsheet, along with profiling information such as the number of instruction cache misses.

3.2 Stochastic Search Algorithm

We base our learning approach to optimization on genetic algorithms. The main motivation is the principle of locality. That is, we believe that that memory mappings with similar parameters are likely to yield similar performance, and thus that better performing mappings are likely to be "close" in the space of possible solutions. It is difficult to derive a simple yet accurate formula to predict the cache performance of a given mapping, making it impractical to use algorithms like gradient descent. Genetic algorithms are ideal for situations where the principle of locality holds and one has an effective way of evaluating the fitness or "goodness" of a solution.

Our stochastic optimization strategy searches the space of possible instruction cache mappings for a given program. To do this, it generates versions of the said program with altered memory alignments. This is done in a typical genetic algorithms fashion. An initial set (population) of modified programs is generated. Then, for each generation, new individuals (new modified programs) are generated, either from scratch or by mating (combining existing solutions). These new programs replace the existing programs whose fitness measurement is lowest, so that the population size remains fixed at each generation. The generational process is repeated as long as desired. In the end, the individual with the highest fitness measurement is chosen as the output of our optimization algorithm.

Genetic algorithms are designed to maximize the average fitness of the population over time. Hence, to optimize instruction cache performance, the fitness value of a given program should ideally be inversely proportional to the number of instruction cache misses the program obtain. Unfortunately, this metric is difficult to obtain in practice. We must also consider that in processors possessing an L2 cache, maximizing the performance of the L1 instruction cache will not necessarily increase the performance. Finally, the number of instruction cache misses a program experiences can vary depending on run-time context, scheduling and I/O. For all these reasons, we chose the running-time as our fitness criterion, which our algorithm tries to minimize over time. Because this is time

consuming, the running-time of any given program is measured only once over a typical input.

Programs with modified memory alignments are generated by modifying the source code of the original program at the assembly level. More specifically, `noop` instructions are inserted after `jump` and `return` instructions. The newly added `noop` instructions are never executed (they are dead code), but change the alignment of the instructions that follow. The resulting programs have a different memory layout, and thus, a different mapping in the instruction cache than that of the original program. The specific locations where dead code is inserted, and how many bytes of code are to be inserted at each location form the genes of a given program. When generating the initial population, the number of genes a specific individual will have as well as the value of each gene are chosen on a uniformly random basis.

The mating of programs is performed by selecting two parent programs from the current population and combining their genes to generate a child program. The parent programs are randomly selected based on a Gaussian distribution, programs with a higher fitness having a higher likelihood of being chosen. Note that we allow the case where both parents are the same program. To combine the genes from both parents, we split the lists of genes from each parent into two halves and merge the first half from the first parent with the second half from the second parent. The points at which each list is split is chosen uniformly randomly. The number of genes of the three programs involved need not be equal, but we do not allow the number of genes of the child to exceed a fixed maximum number. Once the genes of the parents are combined, mutations may be inserted into the new genetic profile. In a typical genetic algorithms fashion, we give each gene a probability of being mutated equal to the inverse of the number of genes. When a gene is selected to be mutated, its new value is randomly re-sampled.

Our algorithm possesses several adjustable parameters, namely the population size, the number of individuals to generate through mating per generation, the number of new individuals to create per generation, the σ parameter of the Gaussian distribution used for mating, the maximum number of genes per individual, and the number of generations to complete. Adjusting these will affect the convergence rate of the algorithm, as well as the solution it converges to. Our design is parametrized based on preliminary experimentation, but full investigation of optimal designs is part of future work. We also note that the nondeterministic nature of this algorithm means it will not always converge to the same solution given the same parameters.

3.3 Cache Coloring Algorithm

To further investigate the potential for i-cache optimization we include an implementation of Hashemi et al.'s *cache coloring* strategy [11]. This algorithm is based on the same core heuristic used in many other i-cache optimizations, and was chosen for its relative simplicity, reliance on only basic profiling data, and easy extension to set-associative designs.

The basic heuristic goal, as discussed in Section 2, is to minimize cache conflicts between immediate caller and callee. From an edge-weighted call-graph (weights are call frequencies), a set of popular procedures and edges are identified. In order of decreasing edge-weight, popular callers and callees are arranged in memory, with heuristic choices to minimize cache conflicts. Full details are of course available in Hashemi et al.'s original paper [11]. Slightly different from their original design, we do not pack in unpopular procedures. We also set the popularity threshold so that the code expansion would be limited and comparable to that of the genetic algorithm. We examine the impact of code expansion through another experiment, discussed below.

As well as a weighted call graph, the algorithm has as input cache size and procedure sizes. Set-associativity is easily accommodated by increasing the allowed number of conflicts for the same cache line when evaluating conflict heuristics, as Hashemi et al. suggest. Our profile data is based on simple `gprof` output, and so does not contain the intra-procedure information required to additionally incorporate procedure splitting; we thus focus on the basic procedure layout problem, treating procedures as atomic units. Once the layout is computed it is given as input to the *layout modifier,* and used to compile a new binary with specific offsets for specified procedures.

3.4 Padding Overhead

The system we have devised to rearrange code involves the insertion of gaps into the compiled code of a program. In the case of the heuristic algorithm explained in section 3.3, these gaps serve to push functions so as to align them with specific cache lines. In the case of our stochastic algorithm, the gaps can be inserted after any jump or return instruction and serve to allow the algorithm to distance portions of code that would otherwise overlap in terms of instruction cache mapping.

In either case, these gaps result in executable binaries that are larger in size than the original program. This could, in theory, negatively impact the performance of the modified binaries in unpredictable ways. To assess this potential impact on performance, we have performed tests on "padded" binaries. These are binaries for which we have moved several functions so that there is as much extra padding data as our heuristic algorithm implementation would have added, but the functions still map to the same cache lines as they originally did. This allows us to assess the impact of the added "bloat" on the running-time of our benchmarks, without directly affecting instruction cache performance.

4 Experimental Analysis

We have tested both optimization strategies on 7 benchmark programs. Six of these (ammp, gcc, mcf, mesa, twolf and wupwise) are from the SPEC CPU2000 benchmark suite and one (radiant) is a custom global illumination 3D rendering program. These benchmarks were chosen because we believe they accurately represent CPU intensive real-world software. Tests were performed on three different platforms (see table 1). Note that the Pentium IV system does not have an instruction cache per se, but rather an instruction trace cache.

Table 1: Description of our benchmarking platforms

Processor	Instruction Cache	L2 Cache	Operating System
Pentium III 733 MHz (Coppermine)	16KB, 4-way assoc. 32 bytes/line	256KB, 8-way assoc. 32 bytes/line	Debian (kernel 2.6)
Pentium IV 2.66 GHz (Northwood)	12K uOps, 8-way assoc.	512KB, 8-way assoc. 64 bytes/line	Debian (kernel 2.6)
Athlon64 X2 3800+ (dual core)	64KB, 2-way assoc. 64 bytes/line	512KB, 8-way assoc. 64 bytes/line	Ubuntu 64-bit (kernel 2.6)

Table 2 shows the number of functions in the compiled binaries for each of our benchmark programs. We have also included a measure of the total code size in bytes, which is a simple sum of the size

of each function. These values are meant to serve as an indication of the complexity of the code in each benchmark. Note that the code sizes reported are for the Athlon64 system.

Table 2: Complexity of our benchmark programs

Benchmark	ammp	gcc	mcf	mesa	radiant	twolf	wupwise
Prog. Language	C	C	C	C	C++	C	Fortran
Function Count	188	1810	35	1021	385	199	32
Total Code Size (b)	130837	1356342	9629	495411	124041	178400	27443
Avg. Func. Size (b)	696	749	275	485	322	896	858

Running time measurements reported in the following sections were obtained by taking an average over 20 runs. Cache miss information was obtained by simulating the cache characteristics using the Cachegrind cache profiler, which is part of the Valgrind tool suite. We have performed tests both without optimizations (O0 compiler flag) and with optimization level 3 (O3 compiler flag).

4.1 Performance

To test our genetic optimization algorithm, we have chosen to perform 40 generations with a population of size 30, 6 individuals created through mating per generation, 3 individuals created randomly per generation, a σ parameter of 0.38 for the mating distribution, and a maximum of 60 genes per individual. The small population size and the small number of generations were chosen primarily to minimize the time needed to complete a genetic optimization cycle, which can take several hours to complete even on our fastest benchmarking platform.

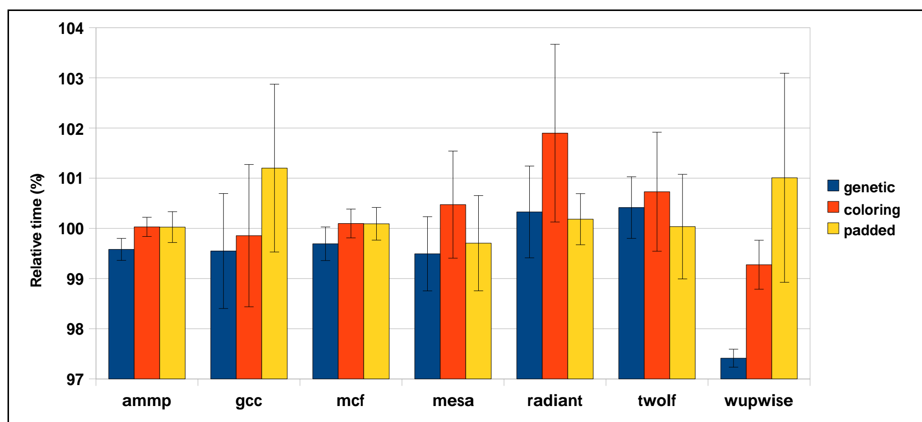

Figure 3: Optimized running times relative to original time (Athlon64, O3)

Comparing the performance gains obtained with the compiler optimizations on and off (see figs. 3 and 4), we see that the picture is quite different. Looking specifically at the mesa and radiant benchmarks, we see that significantly higher performance gains were obtained with the optimizations turned off. We believe there are two reasons for this. The first is that the GNU compilers already do static cache optimizations such as block reordering when optimizations are turned on, leaving less room for improvement. The second is that when optimizations are off, the code generated tends to be larger, and thus, more prone to cache misses, meaning there is more room for optimization.

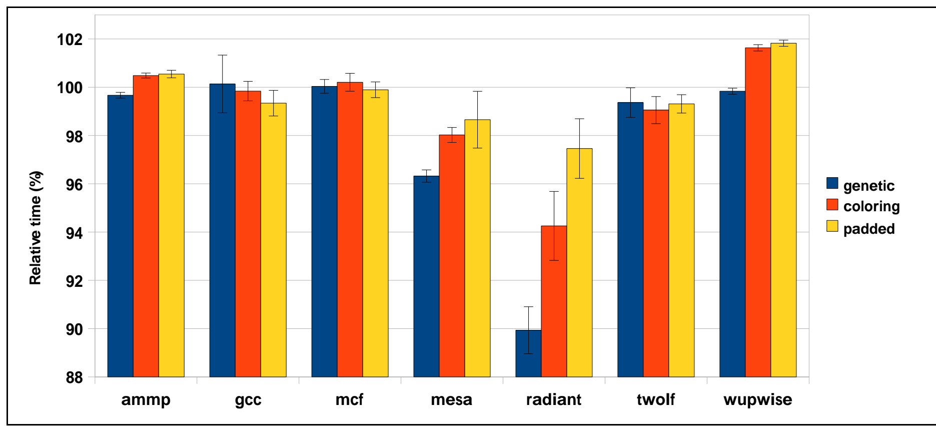

Figure 4: Optimized running times relative to original time (Athlon64, O0)

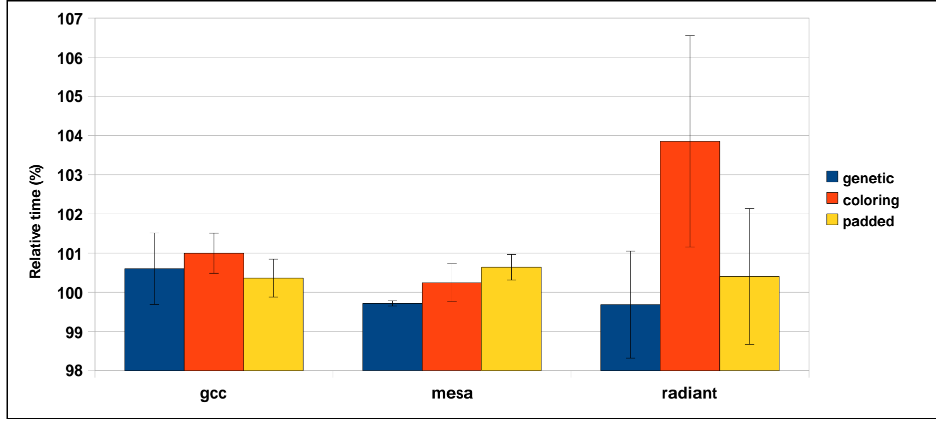

Figure 5: Optimized running times relative to original time (Pentium III, O3)

We have limited our tests to 3 benchmarks on the Pentium III, our slowest platform. This was exclusively due to time limitations, each benchmark needing to be run nearly 400 times per genetic optimization cycle. Coloring is not easily applied in the case of a trace cache, and so these results are not shown on the Pentium IV. The genetic optimization algorithm, however, does not directly require information about cache geometry to run, and produced speedups on both Intel platforms (see figs. 5 and 6).

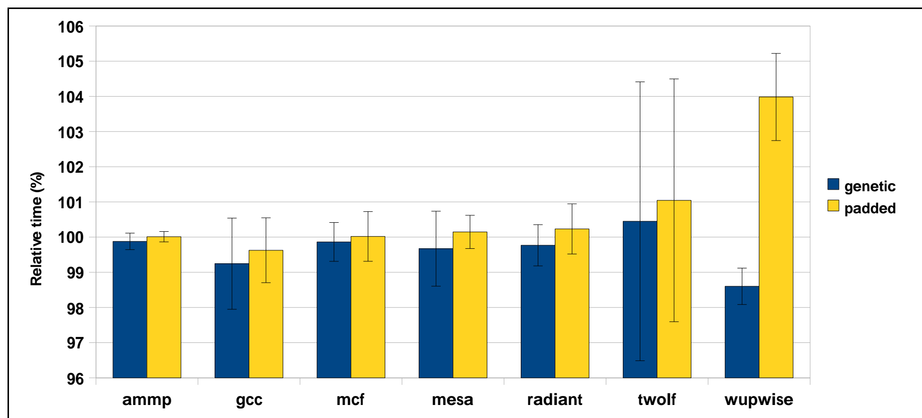

Figure 6: Optimized running times relative to original time (Pentium IV, O3)

In the majority of cases (18 out of 24), our genetic algorithm yields a performance improvement over the original binary, which in some cases is quite significant (see the radiant results in 4). On the other hand, the coloring algorithm, which only does better than our algorithm in two instances, often yields a performance loss. This is not unexpected; since the coloring algorithm is a heuristic, it can easily be wrong, especially if it's heuristic goal does not map directly to speedup. On the other hand, our genetic algorithm actively samples the performance results at every step.

Looking at the relative running time of the padded binaries in figs. 3, 4, 5 and 6, we do not observe a clear trend. In a few cases, the padded binaries perform much worse than the original binaries, however, there are also several cases where the performance of the padded binaries is better. We also see that in the cases where the performance seems to be significantly degraded, the variance is quite high, suggesting that these high running times may not accurately reflect the performance impact of padding.

4.2 Cache Misses

Cache optimization work is typically evaluated in terms of cache performance, with reduced cache misses suggesting a program will improve in speed or cache resource requirements. In the presence of multiple cache levels, however, optimization with respect to one cache level does not necessarily translate into better overall performance. In tables 3 and 4, we present for each of our 7 benchmarks and for both optimization algorithms the relative percentage improvements (positive values are good) obtained in terms of running time on the Athlon64 system with compiler optimization level 3. Along with this, we also show the relative improvements obtained in terms of instruction cache misses and in terms of L2 cache misses for instruction fetches, as well as the approximate number of cache misses before optimization.

We note that our genetic algorithm obtains a 0.5% speed improvement on average, vs. a 0.34%

Table 3: Cache and performance effects of the genetic algorithm (Athlon64, O3)

Benchmark	Time Δ (%)	L1I Δ (%)	L2I Δ (%)	L1I Misses	L2I Misses
ammp	0.42	-33.57	-33.97	0.4M	76K
gcc	0.45	29.95	28.92	15M	3M
mcf	0.31	-7.64	-7.80	1.4K	1.4K
mesa	0.51	-2.34	22.53	1.3M	77K
radiant	-0.33	95.73	-0.58	16M	2.7K
twolf	-0.41	40.73	32.60	33K	28K
wupwise	2.59	-24.12	-25.16	4.7K	4.4K
average	0.50	14.11	2.36	4.7M	0.5M

Table 4: Cache and performance effects of the coloring algorithm (Athlon64, O3)

Benchmark	Time Δ (%)	L1I Δ (%)	L2I Δ (%)	L1I Misses	L2I Misses
ammp	-0.03	-32.11	-68.33	0.4M	76K
gcc	0.14	-8.37	-12.00	15M	3M
mcf	-0.10	-22.45	-22.80	1.4K	1.4K
mesa	-0.47	93.02	15.54	1.3M	77K
radiant	-1.90	93.79	-0.58	16M	2.7K
twolf	-0.73	-60.12	29.92	33K	28K
wupwise	0.72	-12.47	-12.69	4.7K	4.4K
average	-0.34	7.33	-10.13	4.7M	0.5M

speed loss, on average, for the coloring algorithm. What is most interesting is that both algorithms succeed at reducing the L1 instruction cache misses. However, whereas the genetic algorithm also reduces the L2 instruction misses by 2.36% on average, the coloring algorithm causes an average increase of 10.13% for L2 instruction misses. This is likely due to the fact that the coloring algorithm can only concern itself with one of the two caches. As it tries to reduce the L1 instruction cache misses, it may increase the L2 cache misses. Our algorithm does not directly optimize for the cache, but rather for performance, and so avoids this kind of issue.

Interestingly, we note that, according to our data, an improvement in cache performance does not necessarily correlate with a performance improvement and vice versa. The genetic algorithm, for example, obtains a 2.59% speedup for the wupwise benchmark, while its L1I and L2 cache performance decrease significantly. On the other hand, the coloring algorithm reduces the L1I cache misses on the mesa benchmark by 93.02%, but its performance decreases by 1.90%. We also note that the degree of cache performance improvement obtained by both algorithms seems to be proportional to the number of cache misses before optimization. For example, both algorithms obtain a cache performance decrease on the wupwise benchmark, but this benchmark also has the least cache misses before optimization, while both algorithms improve the L1I cache misses of the radiant benchmark by more than 93% from a very high original count of 15 millions.

4.3 Convergence

One important concern when it comes to genetic algorithms and other machine learning algorithms is that of convergence. Our algorithm has many user-adjustable parameters that can affect the rate of convergence, and whether or not convergence occurs at all. The parameters we have chosen for our tests were based on educated guesses, preliminary experimentation, and experimentation time constraints.

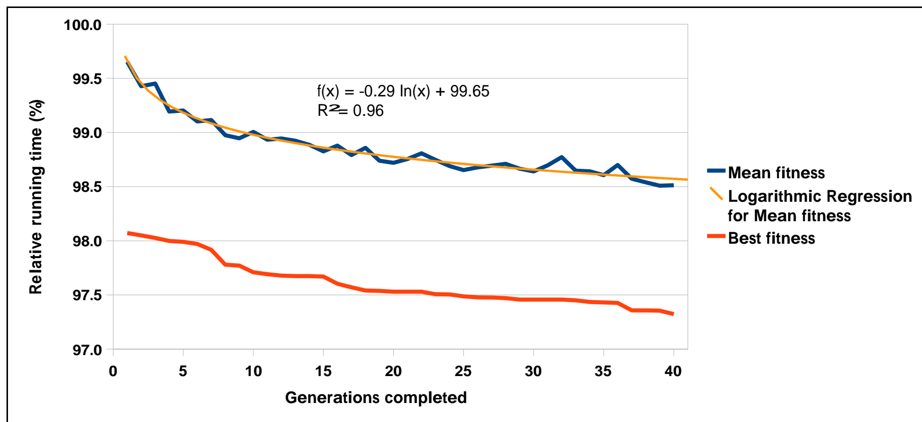

Figure 7: Average relative fitness over time

In figure 7, we examine the relative mean and best fitness (running-time) values averaged over our 24 different benchmark and platform combinations. The mean fitness of the population appears to decrease in an asymptotic, logarithmic manner that is typical of machine learning algorithms. That is, the decrease becomes less and less significant as time passes by because the algorithm is converging to a theoretical maximum performance.

To illustrate the convergence trend, we have performed a logarithmic fit of the mean fitness over time, which we also illustrate. The R^2 coefficient of this fit is 0.96, supporting the idea that convergence is indeed occurring, and that the trend is logarithmic in nature. We can also observe that the best fitness obtained decreases in a similar fashion, and is noticeably lower than the average fitness of the population.

5 Conclusions and Future Work

For some benchmarks, instruction cache optimization has a significant effect, even in the context of other, aggressive optimizations. This performance is not necessarily easy to extract with a simple heuristic and cache model, but as our genetic algorithm optimization shows, useful improvements are possible. Part of the difficulty is due to greater complexity in processor design; multiple cache levels, and other hardware optimization features make simple cache models less heuristically valid. Our basic coloring implementation, for instance, does reduce L1 cache misses, but not in such a way as to generally improve overall performance. A learning approach has the advantage of being able to optimize with respect to execution speed. This allows it to avoid inadvertent performance degradations due to an imprecise or incomplete performance model.

There are many interesting routes for future work. Our framework lends itself to exploration of other algorithms and i-cache optimization designs, and we aim to further investigate other optimization routes. Our implementations, for instance, are constrained by the use of easy to gather profiling and performance data—with more detailed, basic-block level profiling, better results may be possible, both in the learning domain and for deterministic approaches.

Another interesting avenue for future research would be to integrate our stochastic algorithm into a virtual machine, allowing it to optimize code in real-time. This could allow, for example, a service to fine-tune its own performance as it runs. Understanding the benchmarks is also important.

Many other authors have noted a large variation in response to optimization attempts, and the ability to focus effort on only a subset of benchmarks would enable more expensive optimization approaches, such as afforded through machine learning.

References

[1] J. Abella. Near-optimal loop tiling by means of cache miss equations and genetic algorithms. In *ICPPW '02*, page 568, Washington, DC, USA, 2002. IEEE Computer Society.

[2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In *CGO '06*, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society.

[3] S. Bartolini and C. A. Prete. Optimizing instruction cache performance of embedded systems. *Transactions on Embedded Computing Systems*, 4(4):934–965, 2005.

[4] J. P. Bradford and R. Quong. An empirical study on how program layout affects cache miss rates. *SIGMETRICS Perform. Eval. Rev.*, 27(3):28–42, 1999.

[5] J. Cavazos and M. F. P. O'Boyle. Method-specific dynamic compilation using logistic regression. In *OOPSLA '06*, pages 229–240, New York, NY, USA, 2006. ACM.

[6] J. B. Chen and B. D. D. Leupen. Improving instruction locality with just-in-time code layout. In *NT'97: Proceedings of the USENIX Windows NT Workshop 1997*, pages 4–4, Berkeley, CA, USA, 1997. USENIX Association.

[7] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. *SIGOPS Oper. Syst. Rev.*, 32(5):228–239, 1998.

[8] N. Gloy, T. Blackwell, M. D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *MICRO 30*, pages 303–313, Washington, DC, USA, 1997. IEEE Computer Society.

[9] D. Gu, C. Verbrugge, and E. Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE '06*, pages 111–121, New York, NY, USA, June 2006. ACM Press.

[10] C. Guillon, F. Rastello, T. Bidault, and F. Bouchez. Procedure placement using temporal-ordering information: dealing with code size expansion. In *CASES '04*, pages 268–279, New York, NY, USA, 2004. ACM.

[11] A. H. Hashemi, D. R. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. In *PLDI '97*, pages 171–182, New York, NY, USA, 1997. ACM.

[12] X. Huang, S. M. Blackburn, D. Grove, and K. S. McKinley. Fast and efficient partial code reordering: taking advantage of dynamic recompilation. In *ISMM '06*, pages 184–192, New York, NY, USA, 2006. ACM.

[13] J. Kalamatianos, A. Khalafi, D. R. Kaeli, and W. Meleis. Analysis of temporal-based program behavior for improved instruction cache performance. *IEEE Trans. Comput.*, 48(2):168–175, 1999.

[14] K. Pettis and R. C. Hansen. Profile guided code positioning. In *PLDI '90*, pages 16–27, New York, NY, USA, 1990. ACM.

[15] D. J. Scales. Efficient dynamic procedure placement. Technical Report WRL98/5, Western Research Laboratory, August 1998.

[16] K. Sequeira, M. Zaki, B. Szymanski, and C. Carothers. Improving spatial locality of programs via data mining. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 649–654, New York, NY, USA, 2003. ACM.

[17] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *PLDI '03*, pages 77–90, New York, NY, USA, 2003. ACM.

[18] X. Vera, N. Bermudo, J. Llosa, and A. González. A fast and accurate framework to analyze and optimize cache memory behavior. *ACM Trans. Program. Lang. Syst.*, 26(2):263–300, 2004.

[19] D. B. Whalley. Fast instruction cache performance evaluation using compile-time analysis. In *SIGMETRICS '92/PERFORMANCE '92*, pages 13–22, New York, NY, USA, 1992. ACM.