Compiler Optimization Assignment 1

s0840449

February 27, 2012

Abstract

The selection of a correct set of optimizations for an input program is a non-trivial task. The standard approach of preselected sets performs poorly on many programs. One novel approach to the problem is to randomly select optimizations. This paper examines the performance of a single-pass random selection approach, comparing it against GCC's standard optimization sets. Well and poorly performing sets are examined, and a number of optimizations are identified as being influential for the SPEC benchmark.

1 Introduction

Traditionally, compiler developers provide users with manually selected sets of optimizations, such as `-On` ($n = 0, ..., 3$) in *GCC*[1]. However due to the complexity of optimizations and their interactions, choosing a generic set of optimizations is a difficult task. Manually selected optimization sets often do not produce the best code for input programs, and can be detrimental in some cases. This problem has motivated much work in automating optimization selection. Approaches have included optimization-space searching[2, 3, 4], model-based selection[5], and machine learning-based selection[6, 7, 8]. While all of these approaches have shown improved code performance, they require considerable effort to design and implement.

One novel approach that requires almost no work to implement is a random search of the optimization space. In this approach, optimizations are chosen at random on either a per-compiler or per-program basis. This approach has been shown to be surprisingly effective[8, 9, 10, 11], often meeting or exceeding the performance of manually selected optimization sets.

In this paper I examine the effect of applying a single-pass random search of the optimization space on code performance, comparing the results to the standard set of *GCC* optimizations. Additionally, as previous work has suggested that the current GCC optimization sets include optimizations that have at most a minor effect on input code [10], I attempt to use my results to identify optimizations that have a strong beneficial or detrimental effect on code performance.

2 Evaluation Methodology

Evaluations were carried out on *GCC* version 4.4.5. Due to the large number of available optimizations, the optimization space was first reduced to 17 possible optimizations. The reduced space was then randomly sampled 200 times to select optimization sets for evaluation.

A subset of the SPEC2006 benchmark suite[12] was used to evaluate the performance of the random approach. The SPEC benchmarks are used to develop *GCC*'s standard set of optimizations[13], so are a suitable choice for the comparison.

2.1 Optimization Space Reduction

As the optimizations provided in the `-O1` set are generally considered to be 'safe' (always increasing performance), they were chosen as a base for every random selection. The remaining space was then reduced to 17 optimizations; 10 optimizations chosen randomly from the `-O2` set, all 6 optimizations from the `-O3` set, and the `-funroll-loops` optimization. The use of only

17 optimizations reduced the search space size from $2^{137} (\approx 1.74 \times 10^{41})$ to $2^{17} (\approx 1.31 \times 10^{5})$.

The `-funroll-loops` optimization, which is not included in the standard *GCC* optimization sets, attempts to unroll loops where possible. This optimization can improve code performance, but increases the size of the output code and, depending on the size of the loop body, can cause lowered code performance due to instruction cache exceeding.

2.2 Random Search

Once the optimization space had been reduced, 200 optimization sets were chosen using a 'random walk' approach. Each optimization was selected with a $\frac{1}{2}$ chance, giving an even possibility of any set of optimizations being chosen. In the case of the parametrized optimization `-funroll-loops`, the probability for the optimization was divided evenly over five possible parameter values (2, 4, 8, 16, and 32).

3 Experimental Setup

Evaluations were run on a standard DICE machine (*Lisamore*), whose specifications are given in Table 1. Each benchmark was compiled with each of the optimization sets in turn, and was run 10 times per optimization set. The execution time was measured using the `time` command, which measures the elapsed time with millisecond accuracy. The average running time and standard deviation for each set of runs was then computed.

Processor	Core 2 Duo E8400
Processor Speed	3.0 GHz
Processor ISA	x86_64
Memory	4 GB
GCC Version	4.4.5

Table 1: Evaluation machine specifications.

For each benchmark, two groups of optimizations were identified as being 'interesting'. Firstly, any sets of that ran slower than `-O1` were selected, as such sets included optimizations that were reducing code performance. Secondly, the fastest running randomly selected optimization sets were selected; these were defined as either the sets that ran faster than any of the traditional sets, or as the sets that came closest in performance. The optimizations used in each set were then compared in order to identify salient optimizations.

If a set of salient optimizations could be identified, they were either removed (in the case of detrimental optimizations) or isolated (in the case of beneficial optimizations), and the benchmark was re-run to determine how much the identified optimizations effected the code performance.

4 Results

In the results graphs below, the first four columns in each graph are `-O{0,1,2,3}`, followed by the fastest randomly selected optimization set and the average performance of the randomly selected optimizations. Each column gives the average running time, with standard deviations shown as error bars where appropriate. Each graph displays the results from three benchmarks; per-benchmark graphs can be found in Appendix A.

4.1 Bzip2

Bzip2 is an integer benchmark, which uses a version of the `bzip2` library to compress a number of test files. The results for this benchmark can be seen in Figure 1. Of the randomly selected optimization sets, 3 were slower than `-O1`. The salient optimization was `-ftree-vectorize`. Removing this optimization from the slowest set improved its performance above `-O1`.

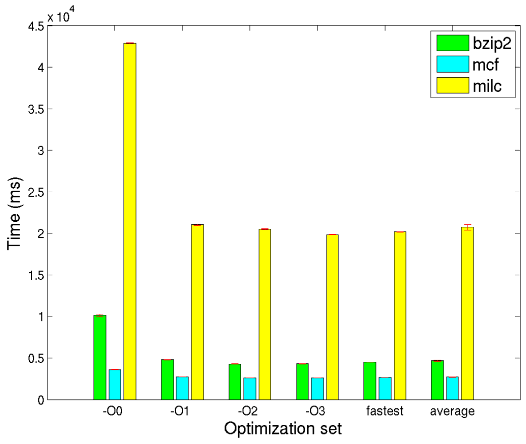

Figure 1: The performances of optimization sets on the bzip2, mcf, and milc benchmarks.

None of the randomly chosen optimization sets performed better then `-O3`. The closest sets were around 6% - 7 optimization sets managed this. The salient optimizations here were `-fcross-jumping`, `-ftree-vrp`, `-fstrict-aliasing`, and `-fstrict-overflow`. Running the benchmark with just these four optimizations gave a performance that was 6.4% slower than `-O3`. The fastest randomly-selected optimization set was '`-O1 --param max-unroll-times=32 -fcrossjumping -fdelete-null-pointer-checks -finline-functions -fipa-cp-clone -fpredictive-commoning -ftree-vrp -fstrict-aliasing -fstrict-overflow -fthread-jumps -ftree-vectorize -funroll-loops -funswitch-loops`', which was 5.4% slower than `-O3`.

4.2 Mcf

Mcf is an integer benchmark, which solves a combinatorial optimization problem. The results for this benchmark can be seen in Figure 1. 30 of the randomly selected optimization sets were slower than `-O1`. The salient optimizations were `-fcross-jumping` and `-funroll-loops` - removing these optimizations from the slowest set improved its performance above `-O1`.

None of the randomly chosen optimization sets performed better than `-O3`. The closest sets were around 3% slower - 58 optimization sets managed this. The salient optimizations here were `-ftree-vrp` and `-fstrict-overflow`. Running the benchmark with just these two optimizations gave a performance that was 3.9% slower than `-O3`. The overall fastest randomly-selected optimization set was '`-O1 -falign-labels -falign-loops -fexpensive-optimizations -ftree-vrp -fipa-cp-clone -fpredictive-commoning -fthread-jumps -ftree-vectorize -fsched-spec -fstrict-aliasing`', which was 3% slower than `-O3`.

4.3 Milc

Milc is a floating-point benchmark, which simulates 4D lattice calculations running on MIMD parallel machines. The results for this benchmark can be seen in Figure 1. Of the randomly selected optimizations, 46 were slower than `-O1`. The salient optimizations were `-funswitch-loops` and `-fthread-jumps` - removing these optimizations from the slowest set improved its performance above `-O1`.

None of the randomly chosen optimization sets performed better than `-O3`. The closest sets were around 2% slower - 18 optimization sets managed this. The salient optimizations here were `-fstrict-aliasing` and `-funroll-loops`, with any `max-unroll-loops` value. Running the benchmark with just these two optimizations and a `max-unroll-times` value of 4 gave a performance that was 2.5% slower than `-O3`. The fastest randomly-selected optimization set was '`-O1 --param max-unroll-times=4 -falign-loops -fdelete-null-pointer-checks`

`-finline-functions -fipa-cp-clone -fpredictive-commoning -ftree-vrp -fstrict-aliasing -fstrict-overflow -fthread-jumps -funroll-loops -funswitch-loops`', which was 1.6% slower than `-O3`.

4.4 Gobmk

Gobmk is an integer benchmark, which analyzes a set of positions in a game of Go. The results for this benchmark can be seen in Figure 2. Of the randomly selected optimizations, 45 were slower than `-O1`. The salient optimization was `-funroll-loops`, with any `max-unroll-times` value. Removing this optimization from the slowest set improved its performance above `-O1`.

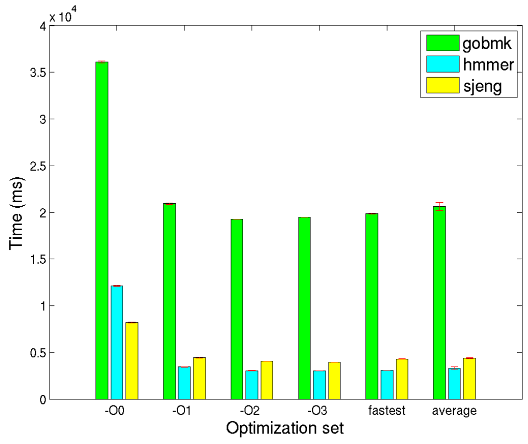

Figure 2: The performances of optimization sets on the gobmk, hmmer, and sjeng benchmarks.

None of the randomly chosen optimization sets performed better than `-O3`. The closest sets were around 4% slower - 15 optimization sets managed this. The salient optimizations here were `-falign-loops`, `-fstrict-aliasing`, `-fstrict-overflow`, and `-ftree-vrp`. Running the benchmark with just these four optimizations gave a performance that was 4.3% slower than `-O3`. The fastest randomly-selected optimization set was '`-O1 -falign-labels -falign-loops -fexpensive-optimizations -ftree-vrp -fipa-cp-clone -fpredictive-commoning -fthread-jumps -ftree-vectorize -fsched-spec -fstrict-aliasing`', which was 3.1% slower than `-O2`.

4.5 Hmmer

Hmmer is an integer benchmark, which performs a statistical search of gene sequences. The results for this benchmark can be seen in Figure 2. Of the randomly selected optimizations, 24 were slower than `-O1`. The salient optimizations were `-fipa-cp-clone` and `-ftree-vrp` - removing these optimizations from the slowest set improved its performance above `-O1`.

None of the randomly chosen optimization sets performed better than `-O3`. The closest sets were around 2% slower - 8 optimization sets managed this. The salient optimizations here were `-fstrict-aliasing`, `-fstrict-overflow`, and `-funroll-loops`, with a `max-unroll-times` value of 4. Running the benchmark with just these three optimizations gave a performance that was 1.6% slower than `-O3`. The fastest randomly-selected optimization set was '`-O1 --param max-unroll-times=32 -falign-labels -falign-loops -fgcse-after-reload -finline-functions -fipa-cp-clone -fpredictive-commoning -fthread-jumps -fstrict-aliasing -fstrict-overflow -funroll-loops -funswitch-loops`', which was 1.3% slower than `-O3`.

4.6 Sjeng

Sjeng is an integer benchmark, which analyzes a chess game. The results for this benchmark can be seen in Figure 2. Of the randomly selected optimizations, 22 were slower than `-O1`. The salient optimization here was `-fcross-jumping`

- removing this optimization from the slowest set improved its performance above `-O1`.

None of the randomly chosen optimization sets performed better than `-O3`. The closest sets were around 9% slower - 5 optimization sets managed this. The salient optimization here was `-funroll-loops`, with a low `max-unroll-times` value. Running the benchmark with only this optimization and a `max-unroll-times` value of 2 gave a performance that was 6.0% slower than `-O3`. The fastest randomly-selected optimization set was '`-O1 --param max-unroll-times=4 -falign-labels -falign-loops -fgcse-after-reload -finline-functions -funroll-loops -fpredictive-commoning -fthread-jumps -ftree-vectorize`', which was 8.6% slower than `-O3`.

4.7 Libquantum

Libquantum is an integer benchmark, which simulates a quantum computer. The results for this benchmark can be seen in Figure 3. None of the randomly selected optimizations were slower than `-O1`, although many were significantly slower than `-O3`.

59 of the randomly selected optimizations sets were faster than `-O3`, with 13 outperforming it by more than 33%. The salient optimizations here were `-fstrict-aliasing` and `-funroll-loops`, with a high `max-unroll-times` value. Running the benchmark with only these optimizations (with a `max-unroll-times` value of 16) gave a performance that was 33% faster than `-O3`. The fastest randomly-selected optimization set was '`-O1 --param max-unroll-times=16 -falign-labels -fcrossjumping -fdelete-null-pointer-checks -finline-functions -funroll-loops -fstrict-aliasing -funswitch-loops -fpredictive-commoning -fgcse-after-reload`', which was 40.6% faster than `-O3`.

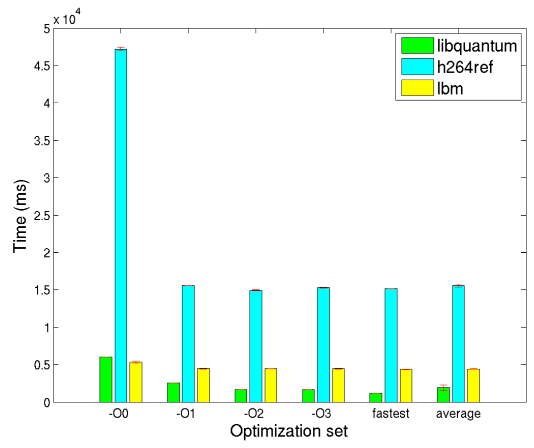

Figure 3: The performances of optimization sets on the libquantum, h264ref, and lbm benchmarks.

4.8 H264ref

H264ref is an integer benchmark, which compresses a set of test video files. The results for this benchmark can be seen in Figure 3. Of the randomly selected optimizations, 57 were slower than `-O1`. The salient optimization was `-funroll-loops`, with any value of `max-unroll-times`. Removing this optimization from the slowest set improved its performance above `-O1`.

None of the randomly chosen optimization sets performed better than `-O2`. The closest sets were around 2% slower - 14 optimization sets managed this. The salient optimizations here were `-strict-aliasing` and `-ftree-vectorize`. Running the benchmark with only these two optimizations gave a performance that was 2.1% slower than `-O2`. The fastest randomly-selected optimization set was '`-O1 -falign-labels -falign-loops -fdelete-null-pointer-checks -ftree-vrp -fstrict-aliasing -fthread-jumps -ftree-vectorize -funswitch-loops`', which was 1.4% slower than `-O2`.

4.9 Lbm

Lbm is a floating-point benchmark, which simulates fluid dynamics. The results for this benchmark can be seen in Figure 3. None of the randomly selected optimizations were slower than `-O1`.

187 of the randomly selected optimization sets performed better than `-O2`, although none outperformed it by more than 2%. Due to the large number of optimization sets, no salient optimizations could be identified. The fastest randomly-selected optimization set was '`-O1 --param max-unroll-times=16 -falign-labels -fcrossjumping -fdelete-null-pointer-checks -fgcse-after-reload -ftree-vrp -finline-functions -fthread-jumps -ftree-vectorize -funroll-loops -funswitch-loops`', which was 1.9% faster than `-O2`.

4.10 Average

Once all the benchmarks had been run, the average performance of each optimization set over all of the benchmarks was computed and used to determine the expected performance of the random selection approach. The average performances can be seen in Figure 4. None of the randomly selected optimization sets were slower than `-O1` on average.

While no randomly selected optimization sets were better than `-O2` or `-O3` on average, 22 were within 2% of their speed. The salient optimizations were `-fstrict-alias` and `-funroll-loops`, with any value of `max-unroll-times`. The overall fastest randomly selected optimization set was '`-O1 --param max-unroll-times=32 -fcrossjumping -fdelete-null-pointer-checks -finline-functions -fipa-cp-clone -fpredictive-commoning -ftree-vrp -fstrict-aliasing -fstrict-overflow -fthread-jumps -ftree-vectorize -funroll-loops -funswitch-loops`', which was 1.4% slower than `-O3` and 1% slower than `-O2`.

The average performance (the expectation) of randomly selected optimization sets on the SPEC benchmark was 4.3% faster than `-O1`, and 6.2% slower than `-O3`.

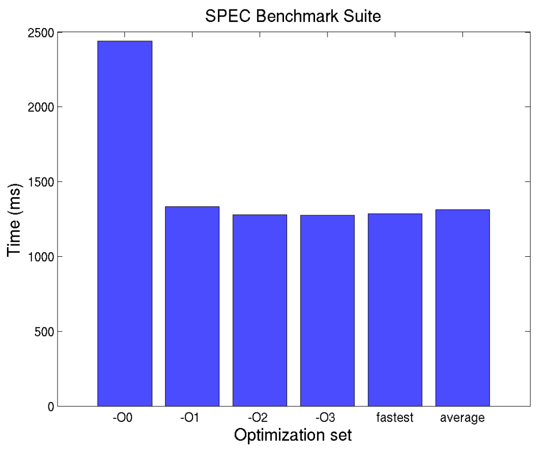

Figure 4: The average performances of optimization sets on the SPEC benchmark suite.

5 Analysis

On average, the single-pass random search approach to compiler optimization selection produced more performant code than the base of `-O1`, but less than the manually selected `-O2` and `-O3` sets. This result is not surprising. The SPEC benchmark suite is used to develop *GCC*, and so it is likely that the `-O2` and `-O3` sets are chosen to be performant on SPEC. Additionally, while reducing the optimization space allows for quicker optimization selection, it may have excluded important optimizations. This can be seen by examining the *Bzip2* or *Sjeng* benchmarks, where none of the randomly selected optimizations could come close to the performance of `-O{2,3}` - it is likely that excluded optimizations are required for good performance on these benchmarks. Finally, the single-pass nature of the approach could account for the poor performance compared to previous results. Most previous work used an iterative approach to compilation, where each input program is compiled multiple times and the performance is used to shape further random searches.

The difficulty of selecting a single good set of optimizations can be seen by examining the fastest average randomly selected set. While this set performed within 1% of the fastest randomly selected set in 6 of the 9 benchmarks, it performed poorly on the *Gobmk* and *H264ref* benchmarks (being 2.5% and 2.9% slower than the fastest sets), and performed extremely poorly on the *Hmmer* benchmark, where it was 6.7% slower than the fastest set. This indicates that per-program optimization selection is preferable to per-compiler, as different optimization sets may perform well on one program but not on another.

The effect of salient optimizations on the SPEC benchmark is noticeable, for both negative and positive optimizations. In all cases, the sets that performed worse than `-O1` needed at most two optimizations removed to run faster than `-O1`. However, there appears to be no negative salient optimization for the whole SPEC suite - most of the negative salient optimizations appeared in only one benchmark each.

It is easier to identify positive salient optimizations for the SPEC benchmark suite. As with the negative case, most of the performance of the fastest optimization sets could be achieved with just a few optimizations. In the case of the *Sjeng* benchmark, the salient set performed better than any other randomly selected optimization set. Overall, two optimizations seemed to be saliently beneficial for the SPEC benchmark suite - `-fstrict-aliasing` and `-fstrict-overflow`.

Two other optimizations, `-funroll-loops` and `-fcrossjumping`, were interesting as they appeared in both the beneficial and detrimental salient optimizations. The `-funroll-loops` optimization appears four times in the positive salient optimizations, but also three times in the negative salient optimizations. This result supports its lack of use in `-O{1,2,3}`, as it is a very program-dependent optimization. A similar situation possibly exists for `-fcrossjumping` (which attempts to unify equivalent code), as it appears once in the positive salient optimizations but also twice in the negative salient optimizations. These cases again support the notion of per-program optimization selection, as well as indicating that it may be important to find approaches that can learn the correct use of salient optimizations for an input program, instead of manually or randomly selecting them.

6 Conclusion

In this paper I investigated a single-pass random search approach to compiler optimization selection, comparing it to the traditional approach of manually selecting optimization sets. Additionally, I attempted to identify salient optimizations that are able to heavily influence the performance of compiled code, either positively or negatively.

I found that the single-pass random approach was able to show some performance over its base of `-O1`, but was not able to match the stronger `-O{2,3}` sets. Two areas for improvement were identified - using a larger optimization search space, and using *iterative compilation*.

Finally, I was able to identify some beneficial salient optimizations for the SPEC benchmark suite, but no detrimental ones. Some optimizations were identified as both negative and positive for different benchmarks, which indicates that an approach that is able to learn good optimizations for each input program may be required for better performance.

References

[1] GCC Optimization Options, `http://gcc.gnu.org/onlinedocs/gcc-4.4.5/gcc/Optimize-Options.html`.

[2] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive Optimizing Compilers for the 21st Century. In *Journal of Supercomputing*, 2003.

[3] B. Franke, M. F. P. O'Boyle, J. Thomson, and G. Fursin. Probabilistic Source-Level Optimisation of Embedded Programs. In *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, 2005.

[4] Z. Pan, and R. Eigenmann. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *International Symposium on Code Generation and Optimization*, 2006.

[5] K. Yotov, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A Comparison of Empirical and Model-Driven Optimization. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2003.

[6] 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*, 1999.

[7] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler Optimization-Space Exploration. In *International Symposium on Code Generation and Optimization*, 2003.

[8] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 185-197, 2007.

[9] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2000.

[10] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Generating New Compiler Optimization Settings. In *Proceedings of the International Conference on Supercomputing*, 2005.

[11] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using Machine Learning to Focus Iterative Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[12] The Spec2006 Benchmark Suite, `http://www.spec.org/cpu2006/`

[13] The GCC Benchmarks, `http://gcc.gnu.org/benchmarks/`

A Graphs

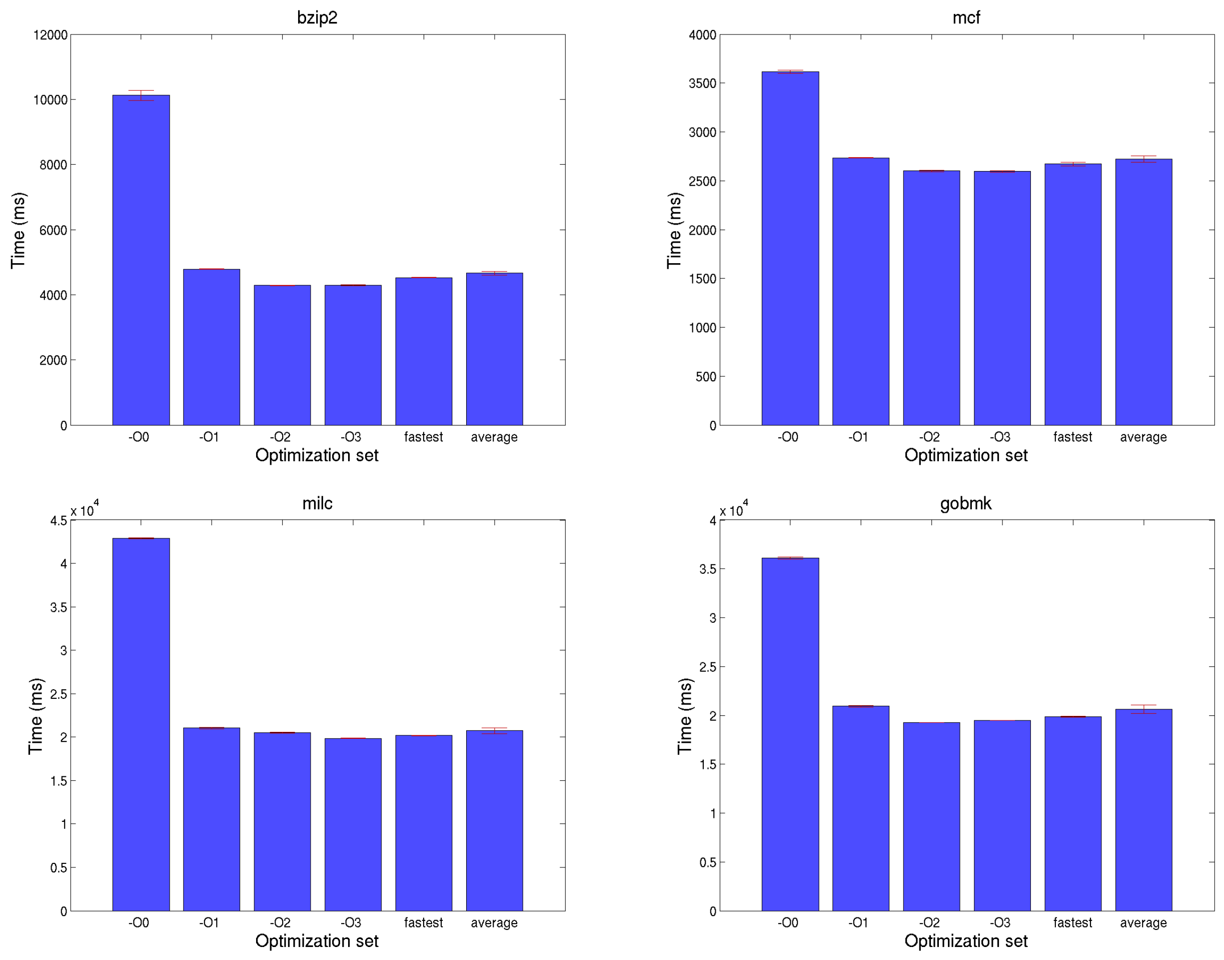

Figure 5: Running times and standard deviations for the bzip, mcf, milc, and gobmk benchmarks.

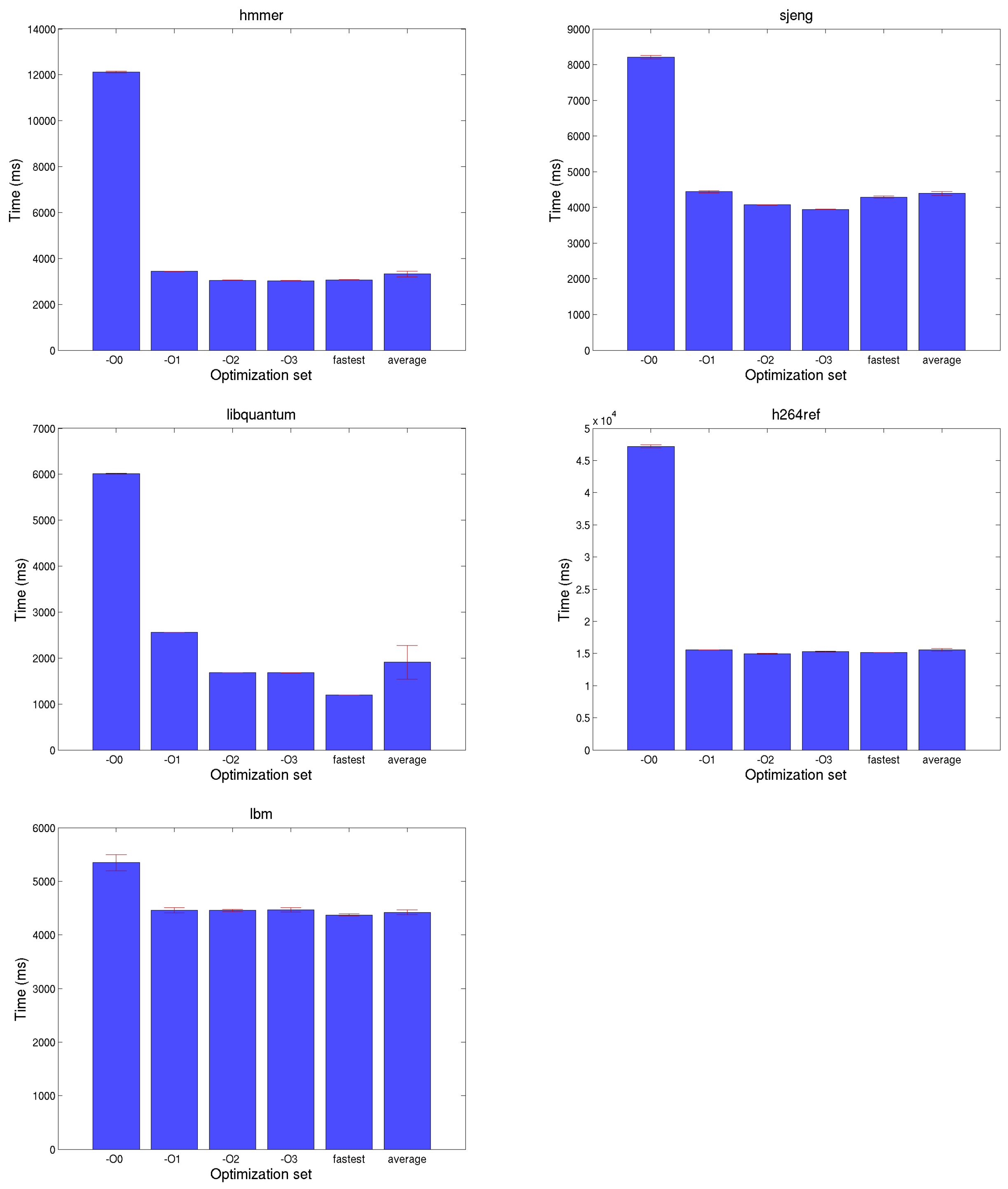

Figure 6: Running times and standard deviations for the hmmer, sjeng, libquantum, h264ref, and lbm benchmarks.