Boosting Inlining Heuristics

Nuno P. Lopes

November 29, 2010

Abstract

Most compilers' algorithms are not optimal, since they often target either NP-hard or even undecidable problems. Therefore, these algorithms require good heuristics in order to solve their problems efficiently and to compute a good solution at the same time. However, deriving good heuristics is a time-consuming task, since each heuristic must be tuned for each particular architecture the compiler aims to support. In this paper we apply statistical techniques (namely boosting) to *automatically* derive a heuristic for inlining.

1 Introduction

Most compilers' algorithms target either NP-hard or even undecidable problems. Therefore, these algorithms are not optimal and rely on heuristics to deliver good results in an efficient manner. However, producing these heuristics is a time-consuming task. Each heuristic must be carefully tuned for each combination of CPU, operating system, memory architecture, cache size, and so on, that the compiler supports.

In this paper we apply techniques from the statistical domain to the problem of *automatically* deriving heuristic functions for compilers. We apply the AdaBoost algorithm [6] to derive a heuristic function for the inlining optimization. This function decides whether a given call site should have its callee inlined or not, and therefore it influences the final binary size and its performance.

Inlining can significantly improve the performance of a program, since it not only removes the overhead of the call instructions sequence, but it may expose opportunities for other optimizations. However, excessive inlining may negatively influence the performance of a program due to a possible increase in instruction cache misses and increased register pressure. It can also result in a bigger binary file (not desirable for embedded systems).

In this paper, we derive two heuristic functions for the inlining compiler optimization: one that optimizes for speed, and one that optimizes for code size. The techniques presented in the paper are directly applicable to other objective functions and to other heuristic functions commonly found in compilers.

2 Related Work

We briefly describe other attempts to apply machine learning techniques to the domain of compiler optimizations.

Dean and Chambers [5] memoize inlining decisions and their impact on the program's running time.

Monsifrot et al. [12] applied a boosting algorithm with decision trees to derive a heuristic for loop unrolling.

Stephenson et al. [14] propose the usage of genetic algorithms to automatically derive heuristic functions for compilers. They generated heuristics for the hyperblock formation, register allocation, and data prefetching algorithms.

Cavazos and Moss [1] used a rule set induction algorithm to derive a heuristic to predict which blocks will benefit the most from instruction scheduling.

Cavazos and O'Boyle [3] applied a genetic algorithm to tune the five parameters of the inliner of the Jikes compiler.

Cavazos et al. [2] used a rule induction algorithm to derive a heuristic to choose which register allocation algorithm should be used in each function.

The MILEPOST project [7] uses genetic algorithms to predict the best optimization ordering for each program. The authors of COLE [8] propose a method based on SPEA2 Pareto optimization to derive the set of optimization passes for each of the optimizations levels (e.g., '-O2', '-O3', '-Os', etc). A recent study [4] showed that tuning the optimizations order yields significant speedups over the manually derived orders shipped in current compilers.

Leather et al. [10] applied genetic algorithms to learn the best set of features, where the feature space is described by a grammar.

Leather et al. [11] describe a methodology to compare the performance of different versions of the same program. Instead of always running the programs for a fixed number of times, they use a statistical approach to decide how many times the programs should be run, since the noise in the measurements can be significant [13].

3 Methodology

Our methodology to derive a heuristic function for inlining consists in the following steps:

1. Compile a set of benchmark programs to the compiler's intermediate representation (IR).

2. Run each program with its input and measure the running time, IR size, and binary size.

3. For each call site in each program, inline the function call, run the benchmarks, and record the running time, IR size, binary size, and the features.

4. Decide for each call site if the inlining was beneficial.

5. Train a heuristic function using AdaBoost and incorporate it in the compiler.

In the following sections we detail each of the points above.

3.1 Compilation of the Benchmarks

We first compile a set of benchmark programs to IR without performing any optimization. We use the clang C/C++ front-end of the LLVM compiler [9] to generate LLVM bitcode. After this step, we have one .bc file for each program that contains all of its code.

3.2 Measuring the Baseline Performance

The second step is to run each program with the training input to measure the baseline performance (i.e., measure the performance of the program without performing inlining). The programs are compiled with the '-O2' set of optimizations, with the exception of the inlining pass that is disabled. We measure the running time, the LLVM bitcode size, and the binary size.

3.3 Inlining Trials

In the third step we compile and run each program several times. One for each call site[1].

In each iteration, we inline a different call site, and we record the value of the features (listed in Appendix A) for that particular call site. We modified the LLVM's inline optimization to do that. The program is then run, and the running time, the LLVM bitcode size, and the binary size are logged together with the features.

This task can be easily parallelizable, since each iteration can be run independently, although we have not done so.

3.4 Categorization

In the fourth step, we have a list of features and measurements, and we need to decide which inlining trials were beneficial. Answering this question depends on the objective: we may want to optimize for speed, we may want to optimize for code size, or a combination of both.

Optimizing for size seems trivial, since we just need to decide to inline when the inlining resulted in a smaller binary. However, since the binaries are padded, small reductions (or increases) in the code size of a function do not impact the final binary size (or even the code section – .text – size). On the other hand, the size of the bitcode is not padded, and thus the changes will be visible there. However, changes in the bitcode size are not directly related with changes in the binary size, due to the long chain of optimizations, instruction selection, and register allocation algorithms and heuristics, through where the bitcode has to pass by.

Optimizing for speed is difficult. We need to run the programs over a set of representative inputs and measure the difference in the running time against the baseline time. As noted in [13], these kind of measurements are very noisy. We chose to run each program for three times and then use the average of the three runs. If the average of the running time is similar with the baseline time ($\pm 0.5\%$), we choose to inline if there was a reduction in the binary size.

[1]We actually limit the number of runs of each program by 500, so that the training step finishes in a reasonable time.

Test Name	# Functions	# Call sites	Bitcode size	Binary size
400.perlbench	1,663	14,725	3,170,988	889,703
401.bzip2	73	330	178,264	57,134
429.mcf	23	78	28,192	17,327
445.gobmk	2,582	9,606	3,759,744	3,530,303
456.hmmer	250	2,402	354,516	135,143
458.sjeng	124	1,238	280,092	123,812
462.libquantum	79	354	35,400	24,450
464.h264ref	462	3,395	1,390,112	505,273

Table 1: Benchmark programs characteristics: number of functions, number of call sites, and bitcode and binary size without performing inlining (in bytes).

Otherwise, we inline if the running time was smaller than the baseline time, regardless of the bitcode or the binary size.

3.5 Training

The final step consists in generating a heuristic function from the data we collected. We use the AdaBoost algorithm [6] to do the training. After generating the heuristic function, we incorporate it in the compiler.

4 Evaluation

We run two set of experiments: one to optimize the code for speed, and one to optimize for code size. The objective was to check whether the proposed methodology produces better inlining heuristics that those found in the compilers for these two common usage scenarios.

4.1 Setup

We implemented our experiments in the LLVM compiler [9] 2.8-svn (r107138) with the clang front-end, and we used the ENTOOL MATLAB classification toolbox[2]. We used the AdaBoost algorithm [6] (40 iterations) for learning.

We used the integer SPEC CPU 2006 benchmarks (only those in C) for training and for the evaluation. The 403.gcc benchmark was excluded from our tests because, at time of writing, clang was miscompiling it. The characteristics of the benchmark programs are shown in Table 1.

The training was done with the 'train' dataset, and the final benchmarking was done with the 'ref' dataset. We limited the number of inline trials to 500 call sites per program for training. Each benchmark/training run was performed three times, and we considered the average of the running times.

The benchmarks were run in a machine with an Intel Core 2 Duo 3.0 GHz CPU and with 4 GB of RAM, running the linux kernel 2.6.33.

[2]Available from `http://www.j-wichard.de/entool/`.

Test Name	No inline	LLVM heur.	Our heur.	Improvement	
				No inline	LLVM
400.perlbench	535	532	534	0.2%	-0.5%
401.bzip2	677	689	676	0.2%	2.0%
429.mcf	374	361	367	2.1%	-1.5%
445.gobmk	579	574	564	2.5%	1.7%
456.hmmer	843	842	846	-0.3%	-0.5%
458.sjeng	653	644	656	-0.5%	-1.9%
462.libquantum	1061	1061	1057	0.4%	0.4%
464.h264ref	852	843	874	-2.6%	-3.6%
Average				0.2%	-0.5%

Table 2: Running time of the programs without performing inlining, and performing inlining with the LLVM heuristic and with our heuristic (in seconds), plus the percentage of improvement of the running time with our heuristic over the no inlining and the LLVM heuristic builds.

4.2 Optimizing for Speed

The results for the heuristic that optimizes for speed are shown in Table 2. Our heuristic improves the performance over the build with no inlining in five tests, achieving 0.2% of average performance improvement, while LLVM's heuristic achieves 0.7%. When compared with LLVM's heuristic, we improve the running time of three tests (including 401.bzip2, which is the only test where LLVM degrades the performance). However, we degrade the performance of five tests, achieving an average decrease of 0.5%. This means that LLVM's current inlining heuristic is better in average than our automatically generated one.

In terms of binary size, the increased performance of LLVM's heuristic comes at the cost of a 17% increase in average, while ours shows only a 2.6% increase in average.

The trained function achieves 94% of accuracy in the training data.

4.3 Optimizing for Code Size

The results for the heuristic that optimizes for code size are shown in Table 3. We can see that the heuristic reduces the code size for five tests, while it does not significantly increase the code size for the other three tests. The average reduction is still 3.3% when compared with compilation with no inlining at all. When comparing with LLVM's '-Os' inlining heuristic, i.e., the heuristic that optimizes for code size, our heuristic still shows an average reduction of 0.9%, although there is a slight increase in code size in four tests.

In terms of performance, our heuristic increases the running time by 0.2%, which can be explained by the fact that we trained the heuristic for aggressive code size reduction without any attempt to improve (or at least do not deteriorate) the performance. LLVM's heuristic reduces the running time by 0.5%.

The trained function achieves 95% of accuracy in the training data.

Test Name	LLVM heuristic		Our heuristic		Reduction	
	bitcode	binary	bitcode	binary	No inline	LLVM
400.perlbench	3,322,928	910,766	3,206,000	886,686	-0.3%	-2.6%
401.bzip2	191,536	58,939	194,780	58,994	3.2%	0.1%
429.mcf	30,820	13,956	28,728	13,967	-19.6%	0.1%
445.gobmk	3,817,472	3,526,082	3,782,132	3,525,929	-0.1%	0.0%
456.hmmer	385,832	139,893	355,728	133,807	-1.0%	-4.4%
458.sjeng	290,420	125,465	288,532	125,763	1.6%	0.2%
462.libquantum	34,240	21,671	31,924	21,913	-10.4%	1.1%
464.h264ref	1,476,076	515,345	1,447,768	506,054	0.2%	-1.8%
Average					-3.3%	-0.9%

Table 3: Bitcode and binary sizes of the programs compiled with the LLVM '-Os' inlining heuristic and with ours, plus the percentage of reduction of the binary size when compared with no inlining and with LLVM's inlining heuristic.

5 Future Work

We leave as future work several improvements to the performance of the proposed method.

First, as each inlining trial is independent of the others, this task can be easily parallelized (provided that a cluster of identical machines is available).

Second, there is a more efficient way to do the inlining trials than inlining just one call site per benchmark run as we currently do. A possible way to do multiple trials per run is to inline one call site per function and then use a profiler or instrument the program to dump the CPU performance counters with function granularity. Then, instead of using the overall running time of the program to decide whether inlining was beneficial, we gather the running time of each function, and thus of each call site (since we only inline at most one call site per function), to make several decisions per run. The number of runs can be minimized if the choice of call sites is cleverly guided by the call graph.

These two improvements combined should allow the number of inlining trials to grow by several orders of magnitude, potentially yielding better heuristics.

6 Conclusions

In this paper, we presented a methodology to derive heuristics for compiler optimizations in a completely automated fashion, which uses the AdaBoost algorithm for learning.

We applied the proposed method to derive two heuristics for the inlining algorithm of the LLVM compiler. One that optimizes for speed, and another that optimizes for code size. The first heuristic improves the running time, but by a smaller amount than LLVM's heuristic. The second heuristic, consistently reduces the binary size, and reduces more in average than when using LLVM's heuristic.

There are three explanations for the bad results achieved by the first heuristic. First, the number of features is small, and some features could be split in multiple features as well. Second, the performance impact of inlining one particular call site is hard to judge, since the improvement or decrease of performance can be easily misclassified as noise in the experiment. Third and last, the recent x86 hardware is not a good target to test inlining algorithms, since the cost of a call instruction is negligible. The performance improvements should be much more noticeable in CPUs without out-of-order execution, VLIW CPUs, or CPUs that do not have the call instruction excessively optimized.

References

[1] J. Cavazos and J. E. B. Moss. Inducing Heuristics To Decide Whether To Schedule. In *Proc. of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, June 2004.

[2] J. Cavazos, J. E. B. Moss, and M. F. P. O'Boyle. Hybrid Optimizations: Which Optimization Algorithm to Use? In *Proc. of the 15th International Conference on Compiler Construction (CC)*, Mar. 2006.

[3] J. Cavazos and M. F. P. O'Boyle. Automatic Tuning of Inlining Heuristics. In *Proc. of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing*, Nov. 2005.

[4] Y. Chen, Y. Huang, L. Eeckhout, G. Fursin, L. Peng, O. Temam, and C. Wu. Evaluating Iterative Optimization Across 1000 Datasets. In *Proc. of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2010.

[5] J. Dean and C. Chambers. Towards Better Inlining Decision Using Inlining Trials. In *Proc. of the 1994 ACM Conference on LISP and Functional Programming*, June 1994.

[6] Y. Freund and R. E. Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*, 55(1):119–139, Aug. 1997.

[7] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, C. Williams, M. O'Boyle, P. Barnard, E. Ashton, E. Courtois, and F. Bodin. MILEPOST GCC: machine learning based research compiler. In *Proc. of the GCC Developers' Summit*, July 2008.

[8] K. Hoste and L. Eeckhout. COLE: Compiler Optimization Level Exploration. In *Proc. of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Apr. 2008.

[9] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO)*, Mar. 2004.

[10] H. Leather, E. Bonilla, and M. OBoyle. Automatic Feature Generation for Machine Learning Based Optimizing Compilation. In *Proc. of the 18th International Conference on Compiler Construction (CC)*, Mar. 2009.

[11] H. Leather, M. O'Boyle, and B. Worton. Raced Profiles: Efficient Selection of Competing Compiler Optimizations. In *Proc. of the ACM SIGPLAN/SIGBED 2009 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Apr. 2009.

[12] A. Monsifrot, F. Bodin, and R. Quiniou. A Machine Learning Approach to Automatic Production of Compiler Heuristics. In *Proc. of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications (AIMSA)*, Sept. 2002.

[13] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! In *Proc. of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2009.

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

A Feature List

Number	Description	Type
ft0	Number of callee's arguments	integer
ft1	Is the callee a library function that is known to be small?	boolean
ft2	Direct function call?	boolean
ft3	Callee returns?	boolean
ft4	Estimate of the benefit of localizing arguments	integer
ft5	Estimate of the benefit of constant propagation through arguments	integer
ft6	Callee uses setjmp()?	boolean
ft7	Callee calls itself recursively?	boolean
ft8	Callee has indirect branches?	boolean
ft9	Callee uses dynamic stack allocation?	boolean
ft10	Callee's number of instructions	integer
ft11	Callee's number of basic blocks	integer
ft12	Callee's number of function calls	integer
ft13	Callee's number of instructions with vector operands	integer
ft14	Callee's number of return statements	integer
ft15	Callee's number of users	integer
ft16	Callee's calling convention	set
ft17	Callee's linkage type	set
ft18	Caller uses setjmp()?	boolean
ft19	Caller calls itself recursively?	boolean
ft20	Caller has indirect branches?	boolean
ft21	Caller uses dynamic stack allocation?	boolean
ft22	Caller's number of instructions	integer
ft23	Caller's number of basic blocks	integer
ft24	Caller's number of function calls	integer
ft25	Caller's number of instructions with vector operands	integer
ft26	Caller's number of return statements	integer