University "Alexandru Ioan Cuza" of Iaşi
Faculty of Computer Science

TECHNICAL REPORT

Learning to Unroll Loops Optimally

Ştefan Ciobâcă and Liviu Ciortuz

TR 08-03, December 2008

ISSN 1224-9327

Universitatea "Alexandru Ioan Cuza" Iaşi
Facultatea de Informatică
Str. Berthelot 16, 700483-Iaşi, Romania
Tel. +40-232-201090, email: bibl@infoiasi.ro

Learning to Unroll Loops Optimally

Ştefan Ciobâcă
LSV, Ecole Normale Supérieure Cachan
and CNRS
94235 Cachan, France
ciobaca@lsv.ens-cachan.fr

Liviu Ciortuz
Department of Computer Science
"Al. I. Cuza" University of Iasi
700483 Iasi, Romania
ciortuz@info.uaic.ro

Abstract

Every few months new types of hardware are released. Compiler writers face the challenging task of keeping the compiler optimizations up-to-date with the latest in hardware technology. In this paper we apply machine learning techniques to predict the best unroll factors for loops, using GCC and the x86 architecture for our experiments. We show that, depending on the machine learning tools used, we get similar or slightly better performance than the native GCC loop optimizer. Our result shows that machine learning techniques can be used to automatically train the heuristic that computes the unroll factor for loops, saving time for compiler manufacturers and providing better performance in the case of scientific computations.

1 Introduction

Compiler writers develop heuristics to solve the problems that occur during the optimization phases of the compilation process. Finding a heuristic for such a problem is difficult and time consuming for the compiler manufacturer, especially when the compiler must work just as well on a number of different architectures.

Common optimization goals include minimizing code size, maximizing speed and minimizing power consumption. Optimization is highly dependent on the target architecture of the system. What works well on the Itanium might not work so well on the x86.

Among the most commonly used optimizations are:

- *function inlining* consists of replacing the call to a function by the body of the function itself. This usually leads to bigger code, but, depending on the archi-

tecture of the system and if the function is sufficiently small, performance is generally better.

- *local instruction scheduling* consists of permuting instructions in a basic block (a sequence of instructions with one entry point and one exit point) in order to improve pipelining.
- *basic block reordering* consists of rearranging basic blocks in order to reduce the branch cost and instruction cache misses.
- *loop unrolling* consists of replacing the body of a loop by multiple copies of itself, while reducing the number of times the loop is executed. This increases locality and reduces the number of jumps. The number of times the loop body is copied is called the unroll factor.

All of the above optimizations were attacked by using tools such as machine learning or genetic algorithms. In [3], Cavazos and O'Boyle use a genetic algorithm to automatically tune the inlining heuristic of a dynamic compiler. In [2], Cavazos uses machine learning techniques to build flexible instructions schedulers. In [5], Liu et al. use neural networks to reorder basic blocks. In [4], Stephenson and Amarasinghe show how to predict unroll factors using supervised classification.

However, very few articles describe experiments on a mainstream architecture/compiler combination, preferring for example the Itanium architecture or even simulators [2] of other architectures. In this paper, we concentrate on using machine learning techniques to predict the best loop unrolling factors. We use the GCC [6] compiler (version 4.1.2), an x86 machine and the SciMark2 scientific benchmark [7].

In section 2, we present the methodology used to gather experimental data and the algorithms used to learn the best unroll factor. Section 3 presents the timing data we obtained from the experiments.

In section 4 we check how the data we gathered from the SciMark2 benchmark generalizes to a different benchmark [8] for a different language: Fortran. This was possible within our existing experimental framework because of the modular architecture of GCC (loops are unrolled at the same place, both for C and Fortran).

Section 5 concludes the paper and gives possible directions for future research.

2 Methodology and Benchmark

The SciMark2 benchmark was originally designed for testing the performance of different implementations of the Java virtual machine. However, the benchmark has been translated to ANSI C, and can be compiled using GCC. We chose the SciMark2 benchmark because it is freely available, fully self-contained, very easy to compile and it contains scientific code, which is amenable to loop unrolling.

There are five computational kernels in SciMark2:

- *Fast Fourier Transform (FFT)* performs a one-dimensional forward transform of 4096 complex numbers.

- *Jacobi Successive Over-relaxation* exercises typical access patterns in finite difference applications.
- *Monte Carlo integration* approximates the value of π by computing the integral of the quarter circle $y = \sqrt{1 - x^2}$.
- *Sparse matrix multiply* exercises indirect addressing and non-regular memory references by multiplying sparse matrices.
- *Dense LU matrix factorization* computes the LU-factorization of a dense matrix using partial pivoting.

There are two versions of SciMark2: with normal data sizes and large data sizes. For the purpose of this paper, we only used the normal version of the benchmark.

To obtain our training data, we modified GCC to output features for every unrollable[1] loop in the benchmark. There are 25 such loops in the SciMark2 benchmark. The following features were computed for each loop and saved for later use:

Feature Name	Explanation
Loop Type (1 or 2)	There are two ways GCC can unroll loops: if the loop tripcount is known at runtime (mostly *for* loops), GCC can apply a rather smart unrolling algorithm, where the loop exit condition is tested for about n/f times, where n is the loop tripcount and f is the unroll factor. For other loops, the test condition must be replicated with each unroll.
Instruction count	The total number of instructions in the loop. This is the most useful parameter.
Branch count	The number of branches in the loop.
Memory reads	How many memory reads are performed in the loop body.
Memory writes	How many memory writes are performed in the loop body.
Function calls	How many functions calls there are in the loop.

We then computed which one of 1, 2, 4, 8, 16, 32, 64 and 128 was the best unrolling factor for each of the loops. To do this, we measured the benefits in terms of time for each loop and for each unroll factor.

[1] Due to technical constraints, GCC may not be able to unroll certain loops at all.

One of the primary concerns is the preciseness of the collected timings. Timing is not a trivial task because the interference of the benchmark with other processes must be minimized. Instead of instrumenting the code to time how much each loop takes, we propose a new scheme to measure the performance of loop unrolling.

To measure the gain of applying a certain unroll factor to a loop, we time the entire program, and not only the loop itself. This has the disadvantage of being slower[2], but is more precise, because the effect of changing the unroll factor for a loop is measured across the entire program.

All tests were run on a 2.0GHz Pentium M machine with 1GB of RAM and 2MB of L2 cache, running Ubuntu 7.04. All non-critical services were shut down during the test run. Each experiment was run several times to minimize the influence of uncontrollable factors on the results.

We mark each loop with the unrolling factor for which the benchmark performed best. On the resulting data, we apply the following machine learning algorithms:

- *DT 2:* decision tree with two of the six features collected (number of instructions and number of branches)
- *DT 6:* decision tree with all six features
- *NN:* nearest neighbor with all six features

To train the decision tree classifiers, we used the J4.8 algorithm, implemented in Weka [1]. We incorporated the classifiers into the GCC compiler (overriding the heuristic that GCC uses to calculate the unroll factor). We then measured the effect of unrolling the loop in terms of speed.

3 Results

Figure 3 summarizes the results in terms of Mflops[3]. When compiled with no unrolling whatsoever, the benchmark achieves 486.03 Mflops. When incorporating a random predictor (which chooses randomly and uniformly the unroll factor from the set $\{1, 2, 4, 8, 16, 32, 64, 128\}$, the score achieved is 506.04 Mflops[4]. The perfect classifier yields a score of 552.61 Mflops. This is thus the theoretical upper bound we can hope to achieve using the data we collected.

Although during leave one out cross-validation, only 28% of the loop unroll factors were classified correctly by the DT 2 classifier, it achieves a score of 528.84 Mflops, better than GCC's built-in heuristic, which achieves 519.98 Mflops.

When using all six features, the decision tree classifier yields a score of 537.35 Mflops, 3.34% better than GCC's heuristic.

For the NN classifier, we only take into account loops for which the program performs significantly better when the loop is compiled with the best unroll factor (versus

[2] Slower in the sense that for each loop and for each possible loop unroll factor, we have to time the entire program.

[3] Millions of floating point operations per second.

[4] The reason for this relatively important increase is that unrolling is generally beneficial for scientific benchmarks.

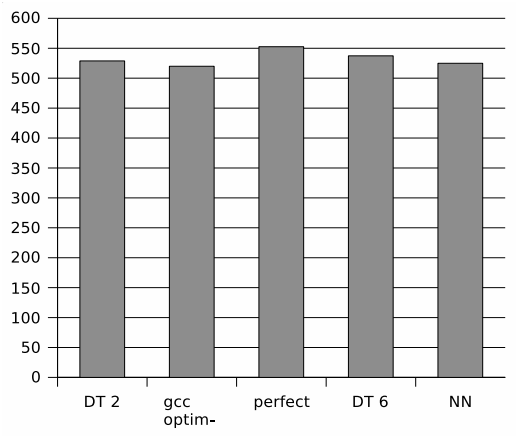

Figure 1: Performance of the different predictors, measured in Mflops over the entire benchmark suite (bigger is better). DT n represents the decision tree classifier with n features. NN is the nearest neighbor classifier.

being compiled with the worst unroll factor). More exactly, we only keep loops for which this difference in performance is greater than 2.5 Mflops[5].

The resulting classifier, which obviously over-fits the data, is very close to the perfect classifier, yielding a score of 549.92 Mflops (the perfect classifier had 552.61 Mflops). To minimize the effect of over-fitting, when we compute the nearest neighbor, we do not take into account the loop itself. This is very similar to leave-one-out cross-validation. The resulting classifier obtains a score of 524.92 Mflops, slightly better then GCC, but slower than both of the decision tree classifiers.

4 Generalization to other Benchmarks and Languages

To verify that our classifiers are valid, we proceed to check how they perform on a different benchmark. We use the Polyhedron 2005 Fortran Benchmark [8] to test if the classifiers generalize to a different language and to a different benchmark.

Of the three classifiers that we trained using the SciMark2 benchmark, the DT 2 classifier performed best on the Polyhedron benchmark, achieving an average speedup of 5.77% over no unrolling at all. However, the GCC loop unroller performed slightly better, achieving a speed up of 6.63%.

Our results show that all three classifiers work well when used on this benchmark, but are not quite as good as the heuristic GCC uses. Table 4 shows the average running time of the 16 programs that are included in the benchmark, when the loop unrolling factor is computed by the corresponding classifier.

[5]Thus, only the *critical* loops were kept; in our case, 7 such loops remained.

classifier	average running time
gcc heuristic	48.84
no unrolling	52.31
decision tree (two features)	49.29
decision tree (six features)	49.39
nearest neighbor	49.65

Table 1: Average running times in seconds (smaller is better) of the Polyhedron benchmarks, with different loop unroll classifiers.

5 Conclusions and Future Work

Using two different machine learning techniques (decision trees and nearest neighbor), we realized a 1.7% (decision tree with two features), 3.34% (decision tree with six features) and 0.95% (nearest neighbor) speed-up on a known scientific benchmark.

In order to obtain the training data, we used a new approach to benchmarking loops. Instead of instrumenting the code around each loop to measure its performance, we measure the entire program for each possible unroll factor for each loop. This leads to improved accuracy of the training data, at the expense of the time required to collect it.

Furthermore, we tested the improvement obtained on SciMark2 with another industry-standard benchmark. The results suggest that the classifiers built with the first benchmark generalize to a different programming language (Fortran) and to a different benchmark suite quite well, but still not meeting the performance of GCC's native loop unroll optimizer.

It is worth noting that while the speed increases do not seem big, they are similar to the speed-ups obtained in [4]. Also, a modest increase in speed can result in significant savings in equipment and maintenance costs.

Also, using machine learning techniques to learn optimizations is a benefit in itself, since the programmer does not need to spend time hand-optimizing the compiler. This also encourages quick release and quick prototyping of compilers.

The results are encouraging and suggest that future compilers may want to include a machine learning phase during the build of the compilers themselves. This way, architecture experts would only need to select the code features that an optimization depends on. The compiler could then use these features selected by the domain expert and learn how they map to the target objective function. The expert doesn't need to spend time tuning the heuristic by hand.

Future work should consider more benchmarks, such as the SPEC [9] benchmark, and more loop features and determine the effects that training the classifiers on one benchmark has on the running time of the other benchmarks. Many other types of compiler optimizations can be amenable to the machine learning techniques we used in this paper; it would be interesting to check which optimizations can be performed by machine learning algorithms and what degree of automation can be achieved.

References

[1] Ian H. Witten, Eibe Frank, “Data Mining: Practical machine learning tools and techniques”, 2nd Edition, Morgan Kaufmann, San Francisco, 2005.

[2] John Cavazos, *Using Machine Learning to Build Flexible Instruction Schedulers*, Master’s Thesis, Department of Computer Science University of Massachusetts, Amherst, 1997

[3] John Cavazos, Michael O’Boyle, *Automatic Tuning of Inlining Heuristics*, Proceedings of the 2005 ACM/IEEE conference on Supercomputing table of contents, 2005

[4] Mark Stephenson, Saman Amarasinghe, “Predicting Unroll Factors Using Supervised Classification”, Proceedings of the 3rd International Symposium on Code Generation and Optimization, San Jose, California, March, 2005.

[5] Xianhua Liu, Jiyu Zhang, Kun Liang, Yang Yang and Xu Cheng, “Basic-block Reordering Using Neural Networks”, SMART’07

[6] ***, The Official GCC website: *http://gcc.gnu.org/*

[7] ***, SciMark2 Benchmark Website: *http://math.nist.gov/scimark2/index.html*

[8] ***, The Polyhedron 2005 Fortran Benchmarks Website: *http://www.polyhedron.com/*

[9] ***, The SPEC Benchmark Website: *www.spec.org*