

Tuning an Adaptive Compiler

Keith D. Cooper, Tim Harvey, and Jeff Sandoval

Rice University, Houston, TX, 77005, USA
{keith,harv,jasandov}@cs.rice.edu

1 Introduction

Adaptive compilation is a technique for feedback-driven selection of program-specific or procedure-specific sequences of code optimizations [1]. Adaptive compilers find effective compilation sequences using search, machine learning, genetic algorithms, and limited enumeration [1–5]. A growing body of literature has established that no single sequence works well for all codes.

Active research in this area focuses on a number of problems, including finding better search techniques [4, 5], understanding the search spaces [1], and reducing the cost of evaluating distinct sequences [2]. This paper examines how the “best” sequences change when we add a new transformation to the search space. Specifically, we added a loop unroller to our pool of optimizations; it changed the “best” sequences and improved overall performance on a suite of benchmark codes.

1.1 Background

Our adaptive compiler focuses on the application of “scalar” optimizations to a low-level intermediate code [2]. Prior to this work, the compiler used 15 distinct transformations. It had no loop unroller, but included a pass, *peel*, that peels the first iteration out of each inner loop. This pass was intended to prepare the code for loop unswitching.

In our experiments, we noticed that the compiler often chose to apply peel repeatedly. For example, the five best sequences for the code `adpcm_coder` each use peel seven times; each ‘p’ indicates a use of peel.

ppppppczpc	ppppppclpc	ppppppclpd	ppppppclpg	ppppppclpm
------------	------------	------------	------------	------------

Table 1. The five best sequences for `adpcm_coder` without unrolling

Because peel eliminates overhead from the first iteration of a loop, it produces a small improvement on each application. As Table 2 shows, peel is chosen quite often. It accounts for 26% of the passes invoked in the best 1% of sequences for our nine benchmark programs.

Where peeling reduces the overhead for the first iteration of the loop, unrolling reduces the overhead for the entire loop. Thus, a single application of unroll achieves a larger improvement than a single application of peel, or even several applications of peel. To see this effect, we added unroll to the fixed sequence against which we test the adaptive algorithms. With unrolling, we saw up to an 8% speedup; unfortunately, on average, it slowed the code down by 5.5%.

Benchmark	Loop-peel Frequency	Benchmark	Loop-peel Frequency
adpcm_coder	42.8%	adpcm_decoder	21.8%
applu	40.0%	matrix300	13.1%
rkf45	41.5%	seval	15.3%
solve	30.5%	svd	24.0%
tomcatv	4.8%	average	26.0%

Table 2. Frequency of peel in top 1% of sequences, by benchmark.

The variability of results from unrolling make it an excellent candidate for the adaptive compiler. Since a pass is only applied when profitable, we can add the unroller and let the search algorithms discover when to use it. Adding an unroller improved the quality of code that the compiler produced and changed both the best sequences found and the distribution of passes in those sequences.

This paper chronicles our experience adding a loop unroller to our adaptive compiler. We measured the compiler’s behavior with and without the unroller. For sequences of 10 passes, the best sequences with unroll produce code that is 10% faster, on average, than those without unroll. Unroll often improves search efficiency, even though it almost doubles the search space size. The compiler usually finds a solution of a given quality more quickly with unroll than without it.

1.2 Loop Unrolling

Loop unrolling clones the body of a loop some number of times (the unroll factor) and adjusts the iteration count [6]. It reduces the number of induction-variable updates and backward branches in the loop. While unrolling is straightforward at the source-code level, our compiler uses an assembly-like intermediate code, where loops are harder to recognize. In this setting, the implementation of unrolling is more complex. Though the details are beyond the scope of this paper, our method combines cycle analysis in control-flow graphs [7] and induction-variable detection [8] in static-single-assignment graphs [9].

Qasem *et al.* argue that a phase-ordering framework such as ours is a poor place to choose transformation parameters like the unroll factor [3]. Selecting an integer parameter for the unroller poses different challenges than does sequence finding. Furthermore, a different unroll factor might make sense for each loop. To handle this issue, we constrain the behavior of the unroller. When applied, it unrolls every inner loop by a factor of four. The search can achieve larger unroll factors by applying the pass multiple times.

1.3 Hill Climber Framework

We performed this work using the impatient hill climber in our adaptive framework [2]. The hill climber starts at a randomly-selected sequence and examines its Hamming-1 neighbors in random order. If it finds a neighbor that executes fewer operations, it moves to that sequence and begins to examine its neighbors. When it can find no improvement, it declares the sequence a local minimum.

To limit search times, the hill climber is impatient; it examines a limited set of neighbors. If none of those sequences is an improvement, it restarts from a new sequence. Experience shows that the hill climber finds good sequences in five to ten descents [2].

Our previous work used sequences of 10 transformations drawn from a set of 15, a space of $15^{10} = 576,650,390,625$ points. Adding unroll enlarges the space to $16^{10} = 1,099,511,627,776$ points, nearly double the size. Though the new space is a strict superset of the old one, it is difficult to predict the effects of the change. We expected that, for programs with opportunities for unrolling, the hill climber would find better sequences, since nearly half of the sequences in the new space contain unroll. We also expected that the search would find “good” sequences faster, because unrolling takes better advantage of opportunities where peel is profitable. On the other hand, programs for which unroll produces no effect—or negative effects—may produce slower searches that return lower quality sequences if the hill climber wastes time evaluating unprofitable sequences.

2 Experimental Results

To determine how adding a transformation changes the system’s behavior, we performed a series of experiments. All measurements show dynamic instruction counts for a simulated virtual machine. Prior experience suggests that we will see similar results with our SPARC and PowerPC backends. The simulator, however, produces stable, easily measured behavior.

2.1 Unroll in a Fixed Sequence

Figure 1 summarizes the impact of adding unroll to our compiler. For each benchmark, the leftmost bar shows the dynamic instruction count for code compiled with our standard fixed sequence [2]. The other measurements are normalized to the performance of the standard sequence. The second bar indicates the performance achieved by applying unroll before the standard sequence. Unroll improves performance up to 8% on `matrix300` and 2% to 6% on `seval` and `tomcatv`. Unfortunately, it degrades performance up to 24% on `applu`, `rkf45`, `solve`, and `svd`, mainly due to small iteration counts in many inner loops. Both `adpcm_coder` and `adpcm_decoder` show no change due to a subtle naming issue that prevents unroll from finding induction variables. Coalescing copies before unrolling would fix the problem; however, the adaptive search automatically finds the correct enabling transformations.

2.2 Unroll in an Adaptive Hill Climber

The two rightmost bars in Figure 1 show the best sequences found by the hill climber without unroll and with unroll. The hill climber used 20 restarts and examined at most 32 neighbors of each point. The hill climber performs well in the larger search space. Each program, except `solve`, shows improvement of 1% to 25% from the addition of unroll. The average improvement from adding unroll, including the degradation on `solve`, is 10%.

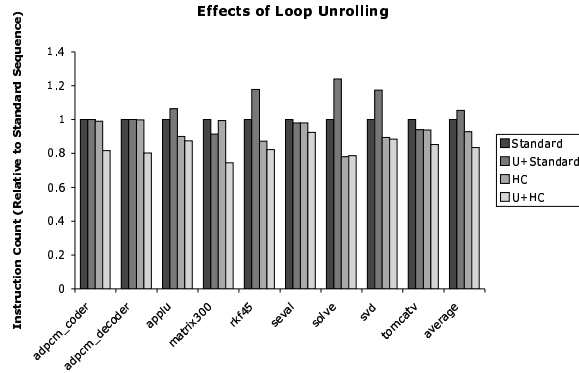


Fig. 1. The effects of adding unroll to a fixed sequence and to the adaptive hill climber. *Standard* refers to the standard fixed sequence and *HC* refers to the hill climber. The *U* modifier indicates the inclusion of unroll for that experiment.

The amount of work required to find a good solution is also critical. We are concerned with search efficiency: performance gain per unit work. Figure 2 shows that the results depend on available opportunity. `adpcm_coder` shows better efficiency with unroll while `solve` shows the opposite result. `rkf45` shows an interesting curve; efficiency with unroll is worse until the search finds a context where unroll pays off; then its efficiency with unroll is better than without it.

Finally, we examined the frequency of peel and unroll in the best sequences from the larger space. Figure 3 displays the average pass frequency for the best 1% of the sequences for each benchmark. Peel is now selected less than half as often as it was before we added unroll. Unroll is selected roughly 11% of the time, showing that the hill climber finds effective uses for it.

3 Conclusion

We improved the performance of code produced by our adaptive compiler up to 25% by addressing a deficiency in its transformation set. Despite the larger search space, the hill climber is usually able to find better compilation sequences with fewer evaluations. These results suggest that the effectiveness of adaptive compilation relies heavily on the capabilities of the underlying optimizations. This kind of evaluation may also lead to a fair basis for comparing transformations.

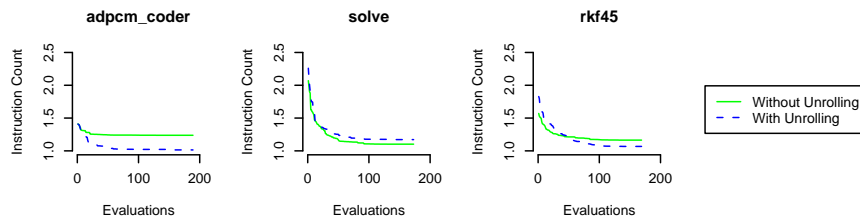


Fig. 2. Search efficiency: average instruction count (normalized against best sequence found) as a function of sequences evaluated

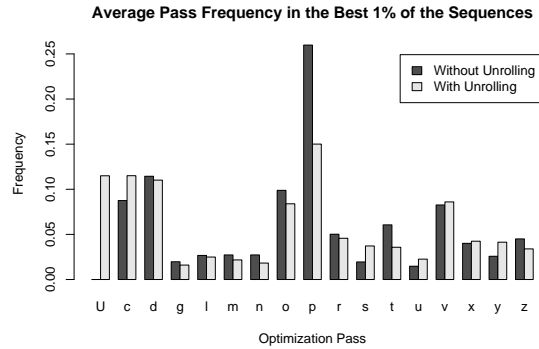


Fig. 3. Average pass frequency in the best 1% of the sequences; p is peel and U is unroll

References

1. Grosul, A.: Adaptive Ordering of Code Transformations in an Optimizing Compiler. PhD thesis, Rice University (2005)
2. Almagor, L., Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S.W., Subramanian, D., Torczon, L., Waterman, T.: Finding effective compilation sequences. In: LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, New York, NY, USA, ACM Press (2004) 231–239
3. Qasem, A., Kennedy, K., Mellor-Crummey, J.: Automatic tuning of whole applications using direct search and a performance-based transformation system. In: Proceedings of the Los Alamos Computer Science Institute 5th Annual Symposium. (2004)
4. Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M.F.P., Thomson, J., Toussaint, M., Williams, C.K.I.: Using machine learning to focus iterative optimization. In: CGO '06: Proceedings of the International Symposium on Code Generation and Optimization, Washington, DC, USA, IEEE Computer Society (2006) 295–305
5. Kulkarni, P., Hines, S., Hiser, J., Whalley, D., Davidson, J., Jones, D.: Fast searches for effective optimization phase sequences. In: PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation, New York, NY, USA, ACM Press (2004) 171–182
6. Allen, F.E., Cocke, J.: A catalogue of optimizing transformations. In Rustin, R., ed.: Design and Optimization of Compilers. Prentice-Hall, Englewood Cliffs, NJ, USA (1972) 1–30
7. Sreedhar, V.C., Gao, G.R., Lee, Y.F.: Identifying loops using dj graphs. ACM Trans. Program. Lang. Syst. **18**(6) (1996) 649–658
8. Cooper, K.D., Simpson, L.T., Vick, C.A.: Operator strength reduction. ACM Trans. Program. Lang. Syst. **23**(5) (2001) 603–625
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4) (1991) 451–490