Procedure Merging with Instruction Caches

Scott McFarling
Western Research Laboratory
Digital Equipment Corporation
100 Hamilton Avenue
Palo Alto, CA 94301
mcfarling@decwrl.dec.com
(415)-853-6651

formerly at Stanford University

Abstract

This paper describes a method of determining which procedures to merge for machines with instruction caches. The method uses profile information, the structure of the program, the cache size, and the cache miss penalty to guide the choice. Optimization for the cache is assumed to follow procedure merging. The method weighs the benefit of removing calls with the increase in the instruction cache miss rate. Better performance is achieved than previous schemes that do not consider the cache. Merging always results in a savings, unlike simpler schemes that can make programs slower once cache effects are considered. The new method also has better performance even when parameters to simpler algorithms are varied to get the best performance.

1 Introduction

This paper presents a method of deciding which procedure calls should be merged to get the best performance on machines with instruction caches. While procedure merging has been studied extensively, the choice of which procedures to merge has remained unclear. In most studies, somewhat arbitrary parameters control the decision. Choosing the right parameter involves a tradeoff between the number of instructions executed and the size of the program after merging. For most recent machine designs, the key cost of increasing program size is the impact on the instruction cache. This cost is reflected in the probability that the next instruction to be executed is not available in the instruction cache or the instruction cache miss rate.

Procedure merging can have both positive and negative effects on the instruction cache miss rate. As more procedures are merged, code size can increase exponentially. The larger programs become, the less likely they are to fit the cache and the higher the miss rate. Alternately, merging can reduce the miss rate. Some procedures may be smaller than the call itself. Merging such procedures reduces both the code size and the miss rate. However, code size is not the only important factor in determining the miss rate. For example, expanding the amount of code that is never executed has no effect on the cache. Also, merging can reduce the miss rate by improving program locality.

Several recent papers have discussed methods of optimizing programs for instruction caches [McF89, HC89a, PH90]. In this paper, optimization for the cache is assumed for two reasons. First, if the instruction cache miss rate is of concern, then programs should be optimized for the cache. Second, a model of the change in miss rate expected from merging particular calls is needed to decide which calls to inline. After optimization, the cache behaves much like an optimal cache. Here, an optimal cache refers to a cache that replaces the instructions used farthest in the future as in Belady's page replacement algorithm [Bel66]. Optimal caches not only have ideal miss rates, they are also relatively simple to model. For example, the placement of instruc-

The MIPS-X project has been supported by the Defense Advanced Research Projects Agency under contract N00014-87-K-0828.

Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.
Toronto, Ontario, Canada, June 26-28, 1991.

tions in memory can be ignored. This paper proposes an accurate model of optimal caches which makes use of profile information and the structure of the program including both the call and flow graphs. This model is then used to guide procedure merging.

Section 2 discusses other work on procedure inlining. Section 3 discusses why the decision of which calls to merge is complex. Section 4 describes the model for predicting miss rates. Section 5 uses the model to predict the benefit of merging any given call. Section 6 uses this to determine the set of calls to merge. Section 7 shows that this new algorithm performs better than simpler methods that do not explicitly consider the cache. Finally, Section 8 summaries the contributions of this paper.

2 Related Work

Numerous compilers implement procedure merging. This section highlights work that discusses the decision of which procedures to merge.

Scheifler [Sch77] studied a method of inlining procedures that used profile information. Calls were inlined in the following order:

1. procedures so small that inlining reduces overall program size.
2. procedures with the highest ratio of dynamic calls to procedure size until an overall program size limit is reached.
3. procedures called from only one site.

Scheifler justifies using heuristics by showing that computing the best procedures to merge is at least NP-complete. The program size limit is arbitrary since the algorithm has no way of trading off program size with the number of instructions executed.

Ball [Bal82] considered how optimization might affect the benefit of doing a merge. Calls with constant arguments may be better to merge since dead code can be removed and arithmetic performed at compile time. Ball also discussed inlining only parts of procedures or creating a specialized version of procedures to be called from several sites with common characteristics.

Richardson and Ganapathi [RG89a, Ric90] compared procedure merging to optimization using interprocedural data flow analysis. They showed that the primary gain from procedure merging is primarily from removal of calls and returns. The savings from merging is largely orthogonal to optimization savings. Using this fact, they propose optimizing before merging to reduce the execution time of optimization.

Davidson and Holler [DH88, DH89] implemented a source to source procedure inliner for the language C. Deeply nested calls are inlined first. Inlining stops when there are no more registers available to hold the variables assigned to registers by the programmer. If user assignments are ignored, inlining can slow the program down without a sophisticated register allocation pass. The authors found that the main benefits from inlining come from the removal of:

- call and return instructions,
- parameter movements,
- stack adjustments, and
- register saves and restores around calls.

Chow [Cho83, Sch83] showed that other optimizations are largely independent of merging for a set of small benchmarks. The savings from both merging and optimization is close to the sum of each performed independently.

Steenkiste [Ste87] implemented a procedure merger that inlined calls to procedures smaller than a size threshold. As the threshold was increased, code size increased roughly exponentially and the number of instructions executed decreased logarithmically. Combining these two effects together, the overall performance went through a maximum improvement of 4%.

Hwu and Chang [HC89b, Cha87] implemented a procedure merger that merged calls in decreasing order of the number of times they were executed until the code size increase reached a threshold. For their set of benchmarks 59% of dynamic calls were removed for a code size increase of 17%.

3 Procedure Call Characteristics

The decision of which calls to merge depends on two factors: the dynamic manner in which procedures are called and the sizes of procedures. If procedures are typically called dynamically from only one place then the decision of which call sites to merge is simple: just merge the site that is frequently executed. Merging this call actually reduces the number of instructions that must be kept in the cache. This is true however many static calls to each procedure there are. Alternately, if most calls are to small procedures, then the choice of which procedures to merge is again simple. Merging small procedures has little effect on program size and thus all calls to small procedures could conceivably be merged. Unfortunately, neither of these simple properties exist in most programs. The set of benchmarks described in Table 1 will be used as examples.

	size (instr)	description
bigfm	8231	graph partitioning
ccal	11526	desk calculator
compare	10112	file comparison
dnf	11411	logic normalization
hopt	16767	compiler optimizer
macro	33689	macro expansion
pasm	12332	assembler
pcomp	35536	pascal compiler
simu	18706	disk simulator
upas	61795	pascal front end

Table 1: Benchmarks Used for Evaluation

Figure 1 shows that most procedures are called frequently from several sites. For each of the benchmarks, the dynamic distribution of calls is plotted against the dynamic fraction of calls to each procedure. Let $c_{j,k}$ be the number of times the k'th call site to procedure j is executed. Let $T_j = \sum_k c_{j,k}$, the total number of dynamic calls to procedure j, and $\mathbf{T} = \sum_{j,k} c_{j,k}$, the total number of dynamic calls. Figure 1 plots the function:

$$y(f) = \frac{1}{\mathbf{T}} \sum_{\{j,k | c_{j,k} \leq f T_j\}} c_{j,k}$$

for each of the benchmarks in Table 1. For any given fraction f, only those calls are included which account for less than or equal to f of the calls to that procedure. For example, if each procedure is called from only one site, then the curve would lie along the x-axis. If each procedure is called frequently from a very large number of sites, then the curve would follow the line $y = 1$. A simple program with one subroutine called the same number of times from two sites would have the distribution:

$$y = \begin{cases} 0 & f < 0.5 \\ 1 & f \geq 0.5 \end{cases}$$

The distributions in Figure 1 vary widely between benchmarks. In one benchmark, 40% of the dynamic calls were the only calls executed to their respective callees. However, most procedures are called frequently from multiple sites. Over all the benchmarks, roughly 40% of calls are from sites that constitute less than half the calls to the callee. This shows that if the majority of dynamic calls are to be removed, many procedures will need to be duplicated and the duplicated code will be frequently executed.

Figure 2 shows the dynamic distribution of callee sizes for the set of ten benchmarks. Most calls are to relatively small procedures with fewer than 100 instructions. However, very few calls are to procedures as small as the call

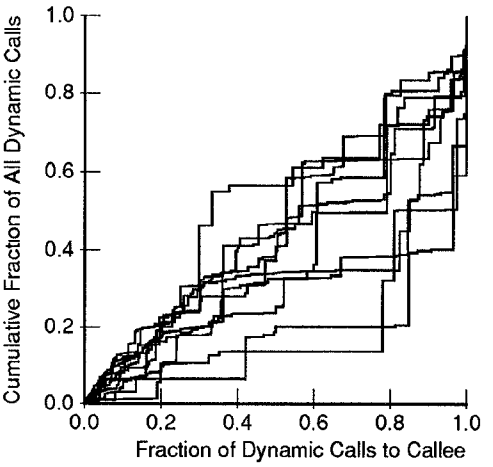

Figure 1: Distribution of Calls to Procedures

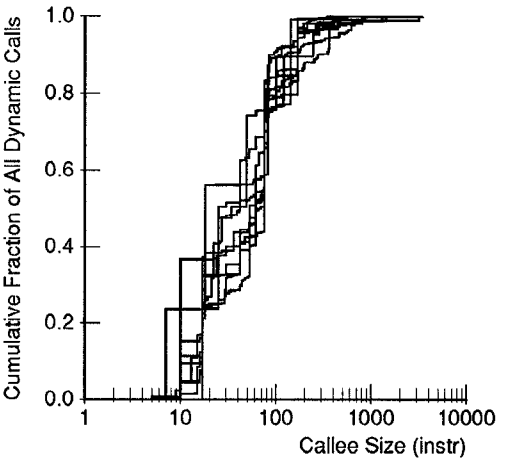

Figure 2: Dynamic Distribution of Callee Size

and return code. In combination with the observation of the previous paragraph, this means that merging will have to increase both the static and dynamic code size to remove the majority of calls. If merging several levels in the call graph is considered, the situation is even more extreme. Code size can increase exponentially. An ideal merging algorithm must consider both the size and frequency of procedures to get the best results.

4 Miss Rate Model for Merging

We begin the description of a new method of determining which procedure calls to merge by presenting a model of how instruction caches behave after optimization. In the model, a single number is calculated for each loop that gives the average size of each loop iteration. Each instruction executed within a loop adds to the average by $min(1, f)$ where f is the average number of times the instruction is executed per loop iteration. When $f \leq 1$,

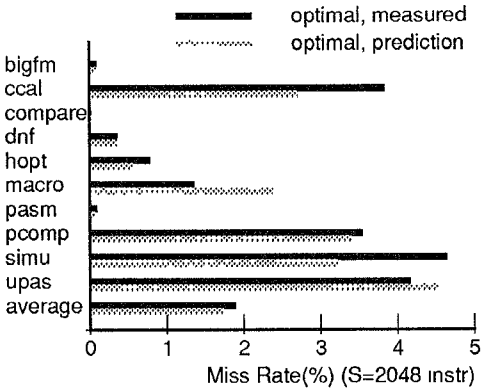

Figure 3: Prediction Accuracy by Benchmark

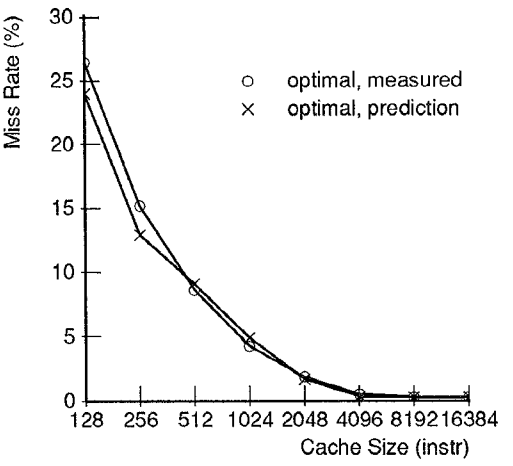

Figure 4: Prediction Accuracy by Cache Size

each instruction adds f to the average loop iteration size. When $f > 1$, each instruction adds only one word to the average because each instruction needs at most one cache location no matter how many times per iteration it is executed.

The average loop iteration size can be used to estimate the number of misses. With cache optimization, each cache location can usually hold one useful instruction just like an optimal cache would. If the loop size is larger than the cache, the surplus instructions miss. If an instruction fits in the cache, it hits when repeated. Whether an instruction misses the first time it is executed in a loop depends on whether there is space to keep in the cache for the duration of the next outer loop. If the current loop is outermost, the first execution always misses. This analysis leads to the estimate of misses for each loop given in Equation 2.

$$s_l = \sum s_b \min(1, f) \quad (1)$$

$$M_l = \max(0, l-1)\max(0, s_l - S) \quad (2)$$

where

s_l	:	effective loop body size
s_b	:	basic block size
f	:	average frequency block is executed per loop iteration
M_l	:	number of misses per loop instance
l	:	average number of loop iterations
S	:	cache size

Figure 3 shows how well the model predicts miss rates for an optimal cache for the set of benchmarks described in Table 1 for a cache size of 2048 instructions. For most benchmarks, the average loop size model is quite close. Figure 4 shows the miss rate prediction accuracy for various cache sizes using the average across the ten benchmarks. Again, the prediction is quite close.

5 Modeling Merge of One Call

Merging a single call affects performance by reducing the number of instructions executed, and by changing the instruction cache miss rate. The change in the number of instructions executed has several causes:

1. removal of call and return code.
2. removal of loads and stores of parameters.
3. opportunities for optimization across procedure boundaries.

Accurate prediction of all these effects is complex. For example, Ball [Bal82] spent considerable effort just calculating the degree inlining procedures with constant parameters enables additional optimization. However, for the MIPS-X architecture [HCe87], the removal of call and return code tends to dominate the other factors. Richardson and Ganapathi [RG89b] along with Davidson and Holler [DH88] found similar results for other architectures. Thus, a simple estimate can be used based on the typical number of instructions eliminated for each removed call. More complex estimates would produce better results, but this simple method is sufficient to demonstrate the usefulness of considering the cache during procedure merging.

To estimate how merging changes the instruction cache miss rate, the loop average size model described in the previous section can be used. Usually, merging a procedure increases code size. To see how this affects the miss rate, the new average sizes of the loops containing the call are computed. The new sizes can then be substituted into Equation 2 to calculate the new number of misses for each loop. If the procedure to be merged is frequently called from only one place, then the average sizes will be reduced by the amount of the call and return code. If the merged procedure is called from multiple

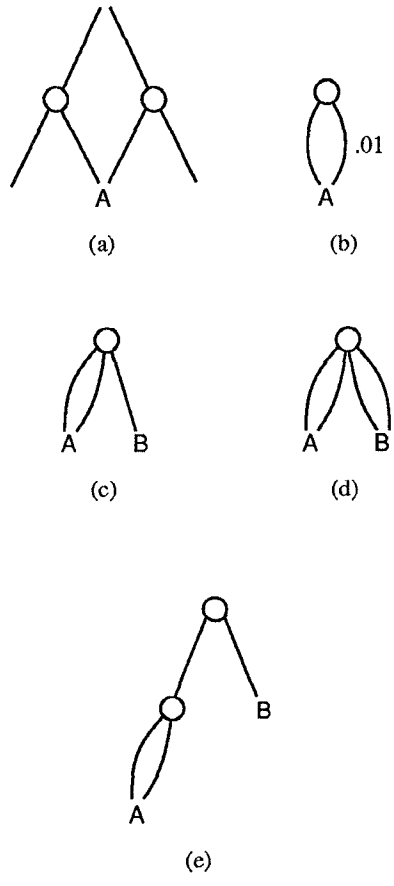

Figure 5: Example Merge Decisions

sites, then some of the loop sizes may increase and the model will predict more misses. Whether a given call should be merged depends on what else is in the loops that contain it.

Figure 5 shows several examples of how these factors can affect whether or not a merge is beneficial. In the figure, circles represent loops, A and B represent procedures, and the arcs represent calls and lexical nesting. Figure 5(a) shows a procedure called only once in each loop it is in. Merging such a call will always increase performance because the number of instructions that need to be kept in the cache over any loop lifetime is decreased. Similarly, merging is always beneficial if there is only one call to a procedure or if only one call is ever executed.

Figure 5(b) shows a procedure called from two places in a loop. One call is executed on every iteration and the other is rarely executed. If both calls are merged, the procedure body from the frequent call will need to be in the cache every iteration. This corresponds closely to the need to keep the original procedure in the cache each iteration. The code from the rare call will demand additional space in the cache, but very infrequently. This corresponds to a small increase in the loop body size as calculated by the model. Unless the called procedure is huge, the reduction in the number of call and return instructions executed will more than compensate for any additional cache misses.

Figure 5(c) shows a loop that contains calls to procedures A and B with B called from one site and A called from two sites. Merging B is always worthwhile since this merge reduces the size of the program. Whether A should be merged depends on the sizes of the procedures relative to the size of the cache. If the loop body will fit in the cache even if procedure A is duplicated, then clearly A should be merged. If A is large and the cache is already full, then A should not be merged since each copied instruction will miss on each loop iteration.

In Figure 5(d), procedures A and B are called from two sites within a loop. Whether or not either procedure should be merged depends on the their sizes and the size of the cache. Since the savings in instructions executed from merging either procedure is approximately the same, it is usually better to merge the calls to the smaller procedure first. Subsequently, if there is still space in the cache, the calls to the other procedure should be considered. As described in Section 6 in more detail, smaller procedures should be considered for merging before larger procedures called the same number of times.

In Figure 5(e), there are two calls to procedure A nested more deeply than a call to procedure B. Whether or not A should be merged depends on the sizes of the procedures and the number of iterations of the inner loop. If procedure B will not fit in the cache if A is copied then the additional misses in B must be weighed against the removal of instructions within the inner loop. This can be calculated using the model from the average number of iterations of the inner loop, the sizes of A and B, and the size of the cache.

6 Deciding What to Merge

The previous section discussed how the benefit of merging a particular call can be calculated. This section describes a method for using this knowledge to decide which combination of calls to merge. The problem is difficult because the decisions for each call interact. Merging one call affects how much space is left in the cache and how often other calls are executed. From the knapsack problem, it can be shown that finding the best set of procedures to merge is at least NP-hard. In addition, merging one procedure causes the calls within the merged procedure to be copied. Deciding whether these new calls should be merged further complicates the problem.

Algorithm MergeDecision in Figure 6 shows a greedy

```
Procedure MergeDecision
begin
    while more calls
        pick aCall with highest (call frequency/callee size)

        savings := aCall.timesCalled * CostOfCall;
        cost := additionalMisses * CostPerMiss;

        if savings < cost then
            Mark aCall to be merged
            update data structures
        end;
    end;
end MergeDecision
```

Figure 6: Merge Decision Algorithm using Loop Model

algorithm for making the decisions. The calls are decided in decreasing order of the frequency of the call divided by the average size of the called procedure. In other words, calls are considered in decreasing order of the expected path length savings versus the amount inlining might increase demand for the cache. Small frequently called procedures are considered first. Each loop that contains the call is analyzed to determine the expected change in the number of misses. The change in misses is then weighted by the miss penalty or cost of each miss and compared with the expected savings in path length to determine whether merging the call will be beneficial. If the model recommends merging the call, the representation of the program is modified to reflect the situation after inlining. If the callee has internal calls that have not been considered for merging, both the original call and the new copy are placed on the undecided list. Any calls already decided keep their current status. Finally, the duplication of the inlined procedure body may create new loops. These loops are analyzed during any future merge decisions.

7 Evaluation

This section shows that using the loop average size model can increase the benefits of procedure merging. Again, the set of ten benchmarks described in Table 1 are used as the basis for the evaluation. For comparison, two simple methods of deciding which procedures to merge are considered. The first does not use profile information at all, just the sizes of procedures. With this method, all calls to procedures with sizes less than a threshold are merged. The second method uses both the size of the callee and the number of times each call site is executed. Calls are merged when the ratio of the number of calls to the size of the procedure exceeds a threshold value. This method is similar to that used by Scheifler [Sch77] except there merging continued until a target code size increase was reached.

Figure 7 shows the average reduction in execution time from merging across the ten benchmarks using both the loop average size model and the size and ratio threshold methods for a range of cache miss penalties. The cache here is direct mapped with the ability to exclude some instructions from the cache, optimized as described in [McF89], and able to hold 2048 instructions. Figures 8 and 9 split the performance change into the path length and miss rate components respectively. The average loop size method uses the miss penalty to decide which procedures to merge. The threshold methods do not. The best threshold to use for a particular miss penalty varies. If the penalty is high, then a threshold that merges fewer calls produces a better result. With the loop average size model, merging is more successful than either threshold method with any threshold value.

Figure 8 shows that as the miss penalty increases, the loop average size model will suggest fewer procedures be merged. As Figure 9 shows, this results in progressively fewer misses as misses get more expensive. With the threshold methods, the miss rate and the path length are independent of the miss penalty. As the miss penalty increases, the improvement with the threshold methods decreases until the programs are actually substantially slower. Figure 9 also shows that unmerged code has a lower miss rate than that of any of the merging decision methods considered here. Of course, the higher miss rate is more than made up for with fewer instructions executed when merging with the loop average size model.

As Table 2 shows, a ratio threshold of 2 gives about the same amount of merging as the loop model with a miss penalty of one cycle in terms of the increase in code size. Figure 8 shows that the resulting savings in instructions executed is also close. However, the miss rate obtained using the cache model is lower. With the procedure size method, a smaller reduction in the number of instructions executed results in a 55% code size increase. The resulting miss rate is substantially higher as well. Merging using the loop model resulted in an increase in code size of only 14%. A small increase is preferable because it minimizes storage costs and the time required to compile the merged code. Again, the primary execution time penalty of increased code size is in the instruction cache and as already shown, the removal of calls more than makes up for this.

Figure 10 compares the loop model improvement to that with a ratio threshold of 2 for a range of cache sizes with a miss penalty of one cycle. The loop model is better across most of the cache size range. While the overall amount of merging is roughly the same, the loop model

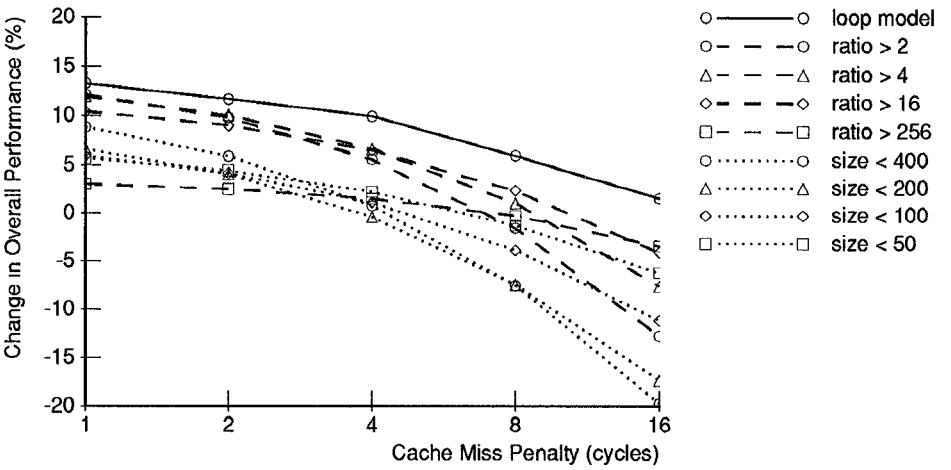

Figure 7: Merging Savings with Increasing Miss Penalty (S=2048 instr)

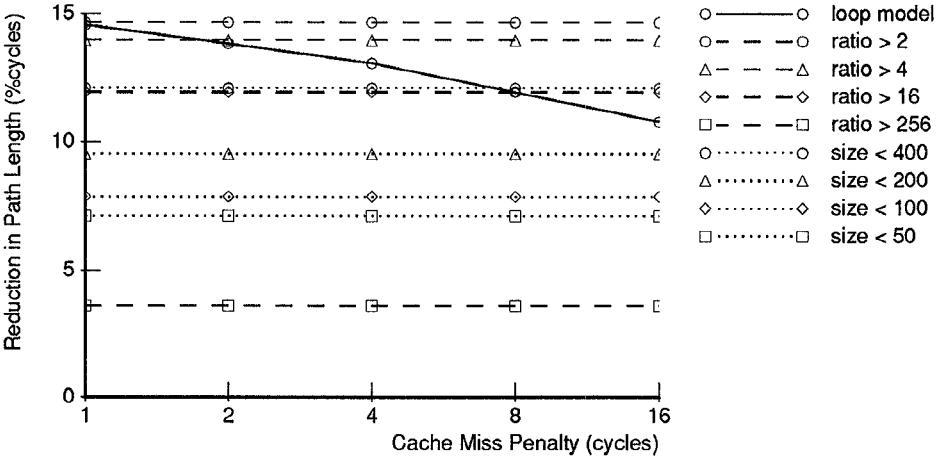

Figure 8: Merging Decrease in Pathlength for Increasing Miss Penalty (S=2048 instr)

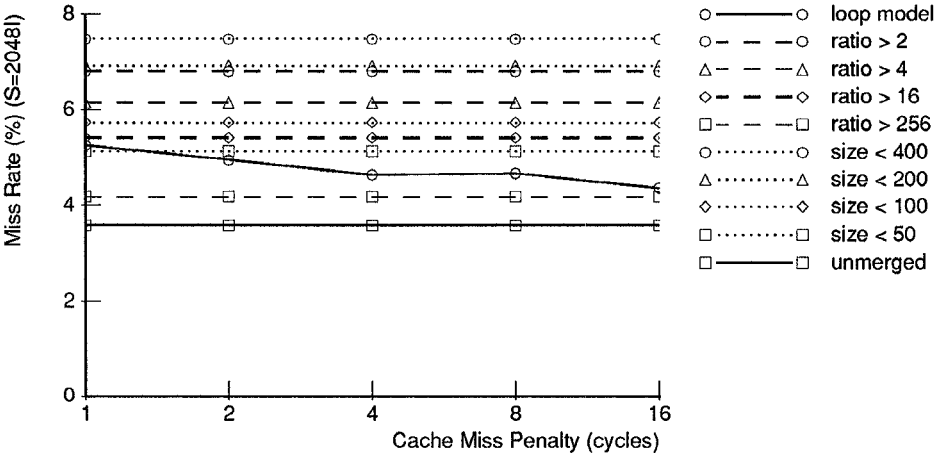

Figure 9: Miss Rate with Merging for Increasing Miss Penalty (S=2048 instr)

	size < 400	ratio > 2	loop model
bigfm	1.29	1.08	1.08
ccal	1.59	1.27	1.32
compare	1.38	1.13	1.18
dnf	1.20	1.07	1.03
hopt	2.15	1.14	1.10
macro	1.07	1.07	1.04
pasm	1.50	1.24	1.23
pcomp	1.57	1.07	1.03
simu	1.96	1.27	1.32
upas	1.77	1.04	1.05
average	1.55	1.14	1.14

Table 2: Relative Code Size Increase (S=2048 instr, miss penalty=1 cycle)

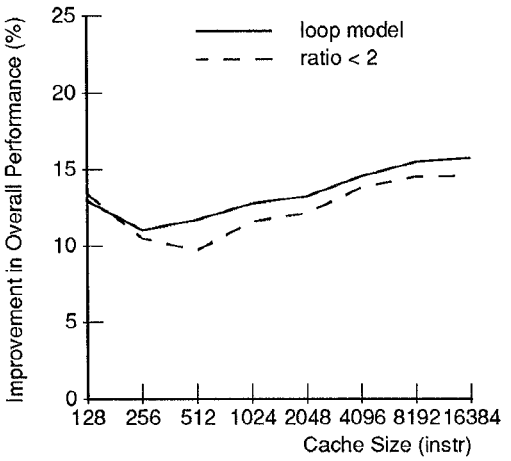

Figure 10: Merging Savings by Cache Size

better finds calls that reduce the number of instructions executed with fewer additional misses. However, for very small caches, the ratio method is slightly better. When miss rates are very high an implicit assumption in the loop model becomes invalid. If a procedure is called multiple times within a loop then an optimal cache will typically keep it in the cache because of its importance. Merging such a procedure results in other procedures being forced out of the cache for loops larger than the cache size. This effect is accounted for in Equation 2. However, if the cache size is very small, even the procedures kept out of the cache may be executed multiple times per iteration. Merging such a procedure increases the size of the loop but does not result in any additional misses since the procedure always misses anyway. Procedure mergers for machines with a very small caches, should add this effect to the model.

Since the loop model uses profile information, it is sensitive to how close the programs behave for different

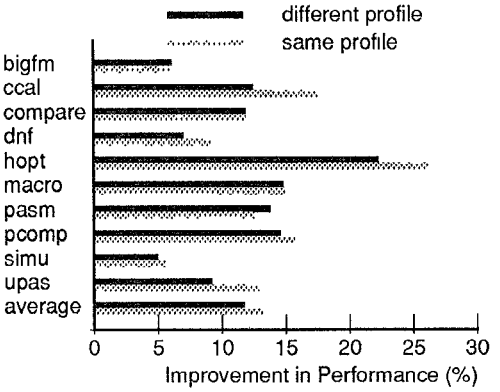

Figure 11: Merge Savings with Different Profile

inputs. Figure 11 shows how the results match when two different sets of inputs for each benchmark are used to guide the merger and each benchmark is run with one of the two inputs. As the figure shows, merging is fairly insensitive to changes in the profile data. Most of the performance gain remains when the profile information comes from a different set of inputs. This is not true where one set of inputs exercised a completely different part of the benchmark than the other. For best results, the profile information for each program should be collected from several runs that accurately reflect how the program is used in practice.

8 Conclusion

This paper has shown that the decision of which calls to merge is complicated because typical programs frequently call large procedures from multiple sites. Thus, if most calls are to be removed, increased code size and additional instruction cache misses are unavoidable. A method of choosing which procedures to merge was given that improves overall performance considering both program path length and the impact on the instruction cache. The technique is preferable to other methods because it does not use ad hoc thresholds, it improves performance across wide ranges of cache sizes and miss penalties, and it produces better performance than algorithms that do not model the instruction cache.

9 Acknowledgments

I would like to thank John Hennessy for several useful comments. I also thank Steve Richardson, Steve Tjiang and the rest of the MIPS-X crew for providing the environment that made this work possible.

References

[Bal82] J. E. Ball. *Program Improvement by the Selective Integration of Procedure Calls.* PhD thesis, University of Rochester, 1982.

[Bel66] L.A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems J.*, 5(2):78–101, 1966.

[Cha87] P. H. Chang. Aggressive code improving techniques based on control flow analysis. Master's thesis, University of Illinois at Urbana-Champaign, 1987.

[Cho83] F. Chow. *A Portable Machine-Independent Global Optimizer - Design and Measurements.* PhD thesis, Stanford University, December 1983.

[DH88] J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software–Practice and Experience*, 18(8):775–790, 1988.

[DH89] J. W. Davidson and A. M. Holler. Subprogram inlining: A study of its effects on program execution time. Technical Report TR-89-04, University of Virginia, November 1989.

[HC89a] W. W. Hwu and P. H. Chang. Achieving high instruction cache performance with an optimizing compiler. In *Proc. 16th Sym. on Computer Architecture*, Israel, May 1989.

[HC89b] W. W. Hwu and P. H. Chang. Inline function expansion for compiling C programs. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 246–257, Portland, Oregon, June 1989. ACM.

[HCe87] M. Horowitz, P. Chow, and et. al. MIPS-X: A 20 MIPS peak, 32-bit microprocessor with on-chip cache. *IEEE Journal of Solid-State Circuits*, SC-22(5):790–799, October 1987.

[McF89] S. McFarling. Program optimization for instruction caches. In *Proceedings of ASPLOS III*, Boston, MA, April 1989.

[PH90] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27, 1990.

[RG89a] S. Richardson and M. Ganapathi. Interprocedural analysis vs. procedure integration. *Information Processing Letters*, 32(3):137–142, August 1989.

[RG89b] S. Richardson and M. Ganapathi. Interprocedural optimization: Experimental results. *Software - Practice and Experience*, 19(2):149–169, February 1989.

[Ric90] S. Richardson. *Evaluating Interprocedural Code Optimization.* PhD thesis, Stanford University, 1990. To appear.

[Sch77] R. W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, Sep. 1977.

[Sch83] D. B. Schnepper. *A Transporter's Guide to to the Stanford U-code Compiler System*, chapter 8. Stanford University, 1983.

[Ste87] P. Steenkiste. *LISP on a Reduced-Instruction-Set Processor: Characterization and Optimization.* PhD thesis, Stanford University, March 1987.