# Program Optimization for Instruction Caches

Scott McFarling
Computer Systems Laboratory
Stanford University

## Abstract

This paper presents an optimization algorithm for reducing instruction cache misses. The algorithm uses profile information to reposition programs in memory so that a direct-mapped cache behaves much like an optimal cache with full associativity and full knowledge of the future. For best results, the cache should have a mechanism for excluding certain instructions designated by the compiler. This paper first presents a reduced form of the algorithm. This form is shown to produce an optimal miss rate for programs without conditionals and with a tree call graph, assuming basic blocks can be reordered at will. If conditionals are allowed, but there are no loops within conditionals, the algorithm does as well as an optimal cache for the worst case execution of the program consistent with the profile information. Next, the algorithm is extended with heuristics for general programs. The effectiveness of these heuristics are demonstrated with empirical results for a set of 10 programs for various cache sizes. The improvement depends on cache size. For a 512 word cache, miss rates for a direct-mapped instruction cache are halved. For an 8K word cache, miss rates fall by over 75%. Over a wide range of cache sizes the algorithm is as effective as increasing the cache size by a factor of 3 times. For 512 words, the algorithm generates only 32% more misses than an optimal cache. Optimized programs on a direct-mapped cache have lower miss rates than unoptimized programs on set-associative caches of the same size.

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

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 0-89791-300-0/89/0004/0183 $1.50

## Introduction

Instruction caches are becoming a more important concern in maximizing overall machine performance. RISC processor speeds have increased much faster than memory speeds. Higher instruction execution rates translate directly into increased demand for instruction bandwidth. Furthermore, since RISC instructions tend to be less dense, additional bandwidth is required. One partial solution is to build separate instruction and data caches[1]. While this doubles bandwidth to memory, instruction bandwidth remains a problem. There is a trade-off between small, high miss rate instruction caches with fast access time and large caches that may increase processor cycle time. For example, the MipsX processor[2, 3] has a small 512 word, 8-way set-associative on-chip instruction cache with a single-cycle access. Misses cause a 20% loss of performance. Even for off-chip caches, the cache needs to be kept reasonably small so that it will fit on the same board as the processor and thus have a fast access time. Additionally, the current trend is for off-chip instruction caches to be direct mapped, even though direct-mapped caches tend to have higher miss rates. Hill[4] and Przybylski[5] show that the increase in access time needed for a set-associative cache usually results in lower overall performance. In summary, instruction cache misses are a problem both for small on-chip set-associative caches and larger off-chip direct-mapped caches.

This paper focuses on improving hit rates in direct-mapped instruction caches using the compiler. Direct-mapped caches have an advantage for optimization because they are more controllable. Most instruction cache misses result from interference between instructions competing for space in the cache. If the compiler finds two instructions that need to be in the cache at the same time, the instructions can be positioned so that they do not map into the same cache block. An additional source of misses is the simple lack of room for some instructions. These instructions not only miss, they may replace useful instructions, potentially doubling the miss rate. There are two solutions to this problem, provided the compiler can discover which instructions are being added to the cache unnecessarily. First, hardware can be added to the cache to exclude instructions that are marked for exclusion. This marking could be done by a bit in the instruction, by address, or by dynamically turning off the cache. Alternately, a normal direct-mapped cache can approach the same behavior if the compiler maps excluded instructions so that they will replace only other excluded instructions.

The remainder of this paper will look at how to discover which instructions are interfering, which instructions should be excluded, and how this information is used to reduce miss rates. Section 3 overviews the entire algorithm. Sections 4 and 5 discuss the labeling algorithm which determines which instructions interfere and which instructions should be excluded. Section 4 describes the labeling algorithm for call graphs that are trees without conditionals, with no loops within conditionals, and with loops nested within conditionals. For the purpose of this paper, conditionals refers to code that is executed on some but not all iterations of its parent loop. Section 5 extends the algorithm to arbitrary call graphs. Section 6 presents an evaluation of the algorithm for 10 large programs. Section 7 presents some concluding remarks.

## Related Work

The problem of optimizing programs for memory system performance has been studied before. Most of the work has focused on reducing page faults of virtual memory machines. Kernighan[6] found an optimal algorithm for inserting page breaks. Hatfield and Gerald[7] used a graph clustering algorithm to group items used at the same time on the same page. Ferrari[8, 9] developed a similar algorithm that modeled how pages would be replaced by a working set algorithm. Baer and Caughey[10] proposed packing programs into pages based on their loop nesting structure. Hartley[11] packed instructions using the call graph. Fabri[12], Abu-Sufrah[13], and Thabit[14] used data flow information to improve memory or data cache performance.

Much less attention has been focused specifically on instruction caches. Chow[15] looked at improving miss rates by sorting procedures by the frequency of call, resulting in very frequent procedures being mapped onto different cache blocks. This helps, but not very much; procedures units are just too large. Substantial portions of a frequently executed procedure may not be executed at all. Moreover, two frequently executed procedures may be called in entirely different phases of execution and be comfortably mapped onto the same cache blocks. Samples[16] used profile information to reduce instruction cache misses by making basic blocks that tend to follow each other in execution also follow in position.

## Algorithm Overview

In this section we describe the overall change to the compilation process. Figure 1 shows the phases of the optimization. The process begins with an object file and a profile file. Using an object file for input has three advantages. First, the program can come from various different compilers or even assembly language. Second, the entire program can be optimized at once. This is necessary because code widely separated among modules can be closely related in time. Third, the sizes of all basic blocks are known exactly. The profile file contains basic block execution counts. There is a trade-off between the benefits of additional information about how a program runs and the cost of obtaining the information. A full instruction address trace would be useful but very expensive to obtain and analyze. As we will see, profile information alone is adequate and easily obtained. Also, simple block counts allow an average to be used if a program tends to have widely different behavior depending on its input.

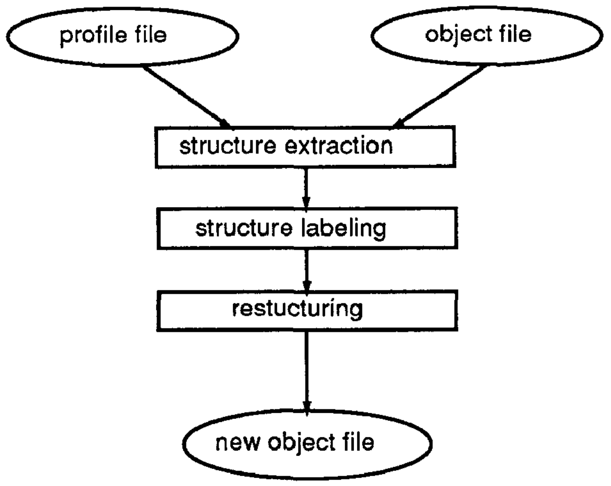


**Figure 1:** Overview of Algorithm

The first phase of the optimization process is structure extraction. The result of this process is a DAG consisting of loop, procedure, and basic block nodes. Loops are labeled with the average number of times the loop is executed (LOOPFREQ). Basic block nodes are labeled with the basic block size. Procedure nodes are the only nodes that can have more than one parent. Arcs are labeled with a fraction that tells how often the child is executed when the parent is executed (ARCFREQ). An irreducible chain of branches is treated as one large loop. Recursive procedure loops are cut and replaced with a normal loop at the head of the recursive loop. For recursive loops with more than one entry point, the cut point is arbitrary.

The details of the structure labeling phase will be discussed in the following sections. For now we just note that the analysis produces two files. One provides a label for some basic blocks (LABEL), the other contains a tree of labels (LABELTREE). Unlabeled blocks can be excluded from the cache. Labeled blocks should be added to the cache. For the purpose of evaluating the effectiveness of the algorithm, we assume that cached instructions are distinguished from uncached instructions by address. Branches are inserted in the restructuring phase to achieve the desired placement. All instructions with the same label and all instructions with descendant labels should be positioned so that they will not interfere in the cache. This is done by placing the root of the tree at cache address zero. The sons of this node are placed so that they all map into the locations immediately following the root node. This continues until all nodes of the tree have been placed in memory. The labeling algorithm attempts to ensure that the height of the label tree is less than the cache size.

## Labeling Algorithm for Trees

In this section we look at the algorithm for labeling instructions. Again, instructions with the same label will be positioned so that they do not interfere and unlabeled instructions will not be added to the cache at all. First, however, we motivate our choice of labels by looking at the behavior of an optimal cache. Belady[17] showed that a paging algorithm that replaces the page that will be used farthest into the future is optimal. For caches the situation is slightly different. It is not necessary to add every instruction to the cache if the machine allows instructions to be passed straight through to the processor. Also, with Belady's algorithm a useless instruction will be added to the cache until something more interesting comes along. We prefer to consider the cache defined below that does not add useless instructions in the first place.

**Definition 1:** A cache is an *OPT* cache if the current instruction is added to the cache, if and only if there is an instruction in the cache that will be referenced later and the current instruction will not be replaced before it is referenced again. If added, the current instruction replaces the instruction in the cache referenced farthest in the future.

For now, let's consider programs where the call graph is a tree, all loops are executed at least 2 times, and there are no conditionals, ie. if a loop is executed, everything in the loop is executed. In this case, the behavior of an optimal cache is quite simple. If the body of a loop is smaller than the cache size, the entire loop will be added. For each instruction inside the loop

```
procedure CombineSons(node)
    INSTR[node] := union INSTR(sons of node)

    for all instructions i in INSTR[node]
        FREQ[node,i] := sum (FREQ[sons of node,i] *
                                    ARCFREQ[arc to son])
    end
end CombineSons

procedure LabelNode(node)
    if node is a leaf
        INSTR[node] := instructions in basic block
        FREQ[node,INSTR[node]] := 1
    else if node is a procedure
        if first visit to node
            LabelNode(son nodes)
            CombineSons(node)
        end
    else (* loop *)
        LabelNode(son nodes)
        CombineSons(node)

        sort INSTR[node] by FREQ with instructions
            with the same FREQ and LABELSET
            grouped together, largest LABELSET first

        INSTR[node] := top cacheSize elements of
                            INSTR[node]

        generate new label loopID
        for all instructions i in INSTR[node]
            if LOOPFREQ[node] * FREQ[node,i] >= 2
            then
                FREQ[node,i] := 2
                add loopID to LABELSET[i]
            else
                FREQ[node,i] := LOOPFREQ[node] *
                                    FREQ[node,i]
            end
        end
    end
end LabelNode

procedure LabelStructure(root)
    LabelNode(root)

    for each unique LABELSET ls
        assign a unique label l(ls)
    end

    for all instructions i
        LABEL[i] := l(LABELSET[i])
    end

    for all unique pairs of LABELSETs ls1,ls2
        if ls1 is a subset of ls2 then
            add an arc <l(ls2),l(ls1)> to LABELDAG
        end
    end

    LABELTREE := Embed(LABELDAG)
        (* for trees Embed is the identity function *)
end LabelStructure
```

**Figure 2:** Algorithm LabelStructure

there must be an instruction in the cache outside the loop that will be used later. If the body of a loop is larger than the cache size, an optimal cache will first add instructions that are required by any inner loops.

The inner loops will replace other code in such a way that as much of them as possible will survive the entire outer loop iteration. Finally, if there is any other space not needed by a inner loop, the first instructions from the outer loop will be added until the cache is full.

> **Theorem 2:** Assume any group of instructions smaller than the cache size can be repositioned sequentially. For programs where the call graph is a tree, every time a loop is executed it is executed at least 2 times, and there are no conditionals, the algorithm LabelStructure in Figure 2 results in the same behavior as the optimal OPT cache.

We can justify Theorem 2 by comparing the behavior of algorithm LabelStructure to the behavior described in the previous paragraph. For small loops, each element of the loop is labeled with its loop id and will be added to cache in such a way that no two instructions in the loop interfere. For large loops, only as much of the loop that will fit in the cache is labeled. For loops with internal loops, code from each son loop in the structure tree is labeled one loop at a time until no more instructions will fit in the cache. This ensures that all instructions labeled with a given loop id will not interfere with any other instruction from that loop with that label.

## Leaf Conditionals

The introduction of conditionals introduces some problems partially because of the limited information that a profile gives us. As an example consider the program structure shown in Figure 3. The figure shows a loop executed 100 times with two basic blocks A and B in the body. Each of the basic blocks are executed on half the loop iterations. Profile information cannot tell us whether the A blocks are all executed first followed by the B blocks ($A^{50}B^{50}$) or whether the A's and B's alternate ($(AB)^{50}$). In the first case A and B do not interfere and can be put in the same locations of the cache. In the second case A and B can not share the same space. Algorithm LabelStructure assumes the sequence is the worst possible and optimizes this case. This may not produce the best possible miss rate for the actual execution sequence but should tend to keep it low.

> **Theorem 3:** Assume any group of instructions smaller than the cache size can be repositioned sequentially and all conditional code is executed the same number of times for each instance of its parent loops. For a program with no loops within conditionals and the call graph is a tree, the worst case miss rate of an optimal cache for all executions of the program consistent with the profile information is the same as the worst case miss rate of the algorithm LabelStructure on a direct-mapped cache with exclusion.

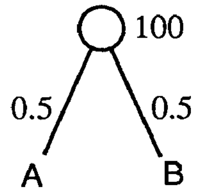


**Figure 3:** Program Structure with Leaf Conditions

> **Definition 4:** The sequence of instructions a program executes is said to be a *Rightmost Instruction Sequence* if for all loops, each child of a loop that is executed on a fraction *f* of the iterations of the loop, it is executed on the last *f* of the iterations.

We can informally prove Theorem 3 by using the above definition and the following three steps:

1. Algorithm LabelStructure caches the same instructions as OPT on the rightmost sequence.
2. The rightmost sequence is a maximal conflict sequence, ie. Miss Rate(OPT(rightmost sequence)) >= Miss Rate(OPT(sequence *s*)) for all sequences *s* consistent with the profile information.
3. Algorithm LabelStructure generates the same number of misses for all sequences consistent with the profile information.

Step 1 can be seen from the way Algorithm LabelStructure prefers the sons of loops with higher frequency (FREQ). If they have higher frequency, they will appear earlier in the rightmost sequence and be cached by OPT.

Step 2 follows since OPT is an optimal cache. It could always cache the instructions it does on the rightmost sequence for any other sequence and get at least as good of miss rate.

Step 3 follows from the fact that instructions added will hit for all but their first reference since they have no competitors for cache space, and all excluded instructions miss independent of which iteration they are executed.

## Loops within Conditionals

Conditionals cause another problem when loops are not always executed for each iteration of an outer loop. For example, in Figure 4 there is a loop containing basic block B that is only executed on fraction q of the iterations of the outer loop. Block B competes for space in the cache with block A. A is executed for fraction p of the iterations of the outer loop. Assuming both A and B are as large as the cache, an optimal cache will always load B since the next block executed will again be B. A will be loaded whenever there are two executions of A without an intervening B. With a direct mapped cache, we don't have this much freedom. We have a choice of either caching both A and B, caching A only, or caching B only. Since caching B will always generate at least as many additional hits as misses, we can exclude the A only choice. To choose between the remaining alternatives we need to know how often A follows A without an intervening B. Unfortunately, profile information cannot tell us. However, if we again assume the rightmost sequence, we can optimize the performance of the worst case execution. If $p>2q$, A and B should be cached. If $p<2q$ only B should be cached. This effect is achieved in algorithm 2 by setting FREQ for instructions that should be cached for the duration of loop equal to 2.

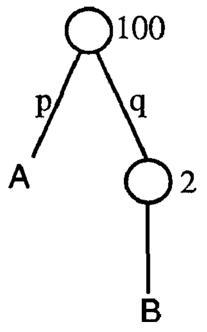


**Figure 4:** Program with Conditionally Executed Loop

## Labeling Algorithm for DAGs

Allowing DAGs into the program structure graph introduces two new problems. First, like loops, DAGs cause the same instructions to be referenced repeatedly. If a small procedure is called twice sequentially, an optimal cache will load it on the first occurrence so that it can be used on the second. Algorithm LabelStructure only partially emulates this behavior. If a procedure is called from more than one place within a given loop, the frequencies of the procedure instructions will be summed so that the procedure will be preferred over instructions accessed less often. This approach may cause more misses than necessary because the order of calls is not considered. For example, in the program in Figure 5, if only one procedure A or B will fit, the algorithm will cache one or the other, whereas an optimal cache will cache both.

```
procedure main;
    for i := 1 to 100 do
        call A;
        call A;
        call A;
        call B;
        call B;
        call B;
    end;
end main;
```

**Figure 5:** Program Illustrating Problems with DAGs

The second problem with DAG program structures occurs because the resulting LABELDAG in the algorithm is in general also a DAG. A DAG does not completely specify how the instructions should be repositioned. With DAGs, it is not possible in general to position the code so that all code labeled with a given loop will not interfere. We would like an algorithm for placing the code to minimize the conflicts. Unfortunately, this problem is NP-complete. We have chosen instead a heuristic greedy algorithm that works quite well in practice.

To motivate the algorithm, consider the example structure shown in Figure 6. Assuming the bodies of each of the loops are small enough to fit in the cache, algorithm LabelStructure will produce the LABELDAG shown in Figure 7. In a LABELDAG, a node interferes with its descendants and nodes with a common descendant for the duration of at least one loop. We would like to position interfering nodes so that they map to different areas of the cache.

Let's first consider the situation if LABELDAG is in fact a tree. We can position the leaf nodes at the bottom of the cache. Successive levels up the tree can be positioned adjacent to their largest son. Since all nodes on a path in the tree are labeled with the same loop id, and there are no more than cache-size instructions assigned to any one loop, the tree can be positioned within the cache size.

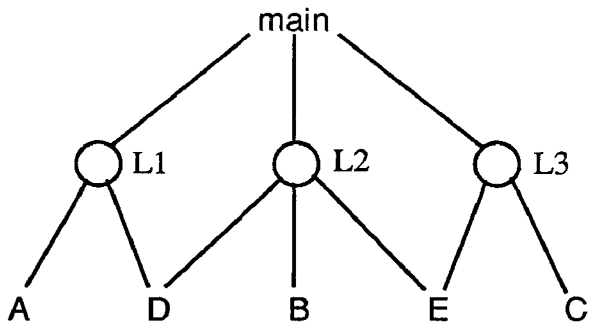


**Figure 6:** Example DAG Structure

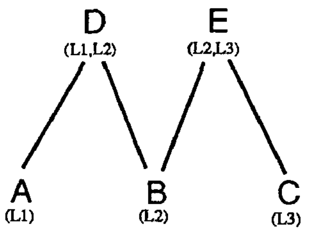
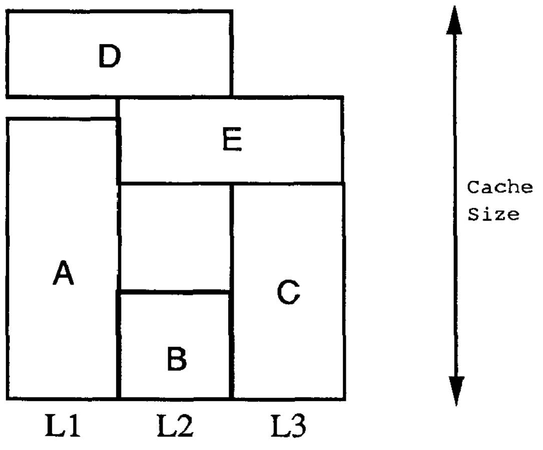
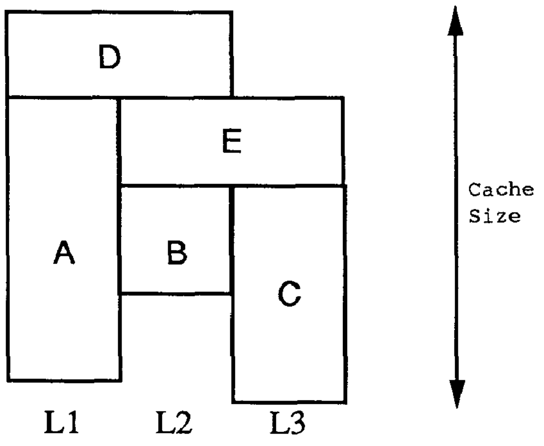


**Figure 7:** Example LABELDAG



(a)



(b)

**Figure 8:** Two Possible Cache Placements

For our example and DAGs in general, the situation is more complicated because nodes with shared descendants interfere. However, we can begin by recognizing that, like trees, all leaf nodes labeled with a single loop id can always share space in the cache. Figure 8(a) shows nodes A,B, and C positioned at the bottom of the cache. In this diagram, the vertical dimension represents the space taken up in the cache, and the horizontal dimension represents the different loop lifetimes or equivalently different addresses that map into the same place in the cache. We have a choice between placing D or E next. In this example, the choice is arbitrary. Figure 9 shows the Embed algorithm which constructs LABELTREE as described above. Embed chooses to place E next because size(A+D+E) > size(C+E+D). This tends to leave as much free space as possible for any parents that still need to be positioned. The algorithm also considers the size of a node's parents. If there are large nodes depending on a node, it is generally good to get it positioned sooner. Figure 8(b) shows A and B moved up in the cache adjacent to D and E. This would be beneficial if it size(D+E+C) were larger than the cache size and D wrapped around onto the bottom of the cache.

```
procedure DOWNSIZE(node)
    return size(node) +
            max(DOWNSIZE(sons of node))
end DOWNSIZE

procedure UPSIZE(node)
    return size(node) + max(UPSIZE(parents of node))
end UPSIZE

procedure Embed(LABELDAG)
    LABELTREE := empty set

    repeat until all nodes in LABELDAG marked
        pick node in LABELDAG with
            1) all sons in LABELDAG marked
            2) max DOWNSIZE(node)
            3) if (2) same, max UPSIZE(node)

        mark node

        add arc from node to son with max DOWNSIZE
            to LABELTREE
    end

    return LABELTREE
end Embed
```

**Figure 9:** DAG Placement Order Algorithm

## Minor Additions

The critical parts of our optimization algorithm have been described. For completeness, we now mention some minor improvements to the basic algorithm not shown in Figures 2 or 9. First, so far we have assumed that if the algorithm selects **n** instructions to be positioned sequentially in the cache, this will only take up **n** cache locations. In practice, this cannot be done. Branches are inserted to preserve the correctness of the program and these branches take up space. This expansion is predicted and the algorithm attempts to only label as many instructions that will fit in each loop.

Another small change concerns code that is included in more than one loop. Each added loop label generates more conflicts in the LABELDAG. Some of these loops are much less important than others based on the frequency the loop is executed. By ignoring the less important loop conflicts, we improve the success of the Embed algorithm in satisfying the important ones.

Finally, code may be selected for inclusion in one loop and exclusion from another. We have to make a choice. As described previously, the algorithm will chose inclusion. The results in the next section use a slightly different algorithm that will exclude code if the loop where it should be excluded is much more important than the loop where it is included.

## Evaluation

In this section we evaluate the performance of the optimization algorithm on a set of ten large benchmarks. We look at how performance compares for a variety of cache organizations and sizes. Table 1 shows the ten benchmarks used in this evaluation. The benchmarks range in size from 8231 to 61705 instruction words. They range from compilers to CAD programs. The programs were compiled and run on a simulator of MipsX that can generate instruction address traces. These traces were then run through a cache simulator.

| | size (words) | description |
|---|---|---|
| bigfm | 8231 | graph partitioning |
| ccal | 11526 | calculator |
| compare | 10112 | file comparison |
| dnf | 11411 | converts logic to normal form |
| hopt | 16767 | compiler optimizer |
| macro | 33689 | macro expansion |
| pasm | 12332 | assembler |
| pcomp | 35536 | pascal compiler |
| simu | 18706 | disk traffic simulator |
| upas | 61795 | pascal compiler front end |

**Table 1:** Benchmarks Used for Evaluation

Table 2 shows miss rates for various 512 word caches on each of the ten benchmarks. Each cache fetches back a single word on each miss. The second column show miss rates for the OPT cache described above with full associativity and look ahead. The third column shows performance achievable with the algorithm presented in this paper for a direct-mapped cache with an uncached address range. The average miss rate is only a third higher than the optimal cache. About half this difference is accounted for by the expansion in code size that occurs with the need to reposition the instructions of a loop sequentially. The code size of the repositioned code increases by about 14%. All branches on MipsX take 3 words of storage, counting delay slots. For machines with smaller branches, the expansion in code size will be reduced. Column 4 shows the results of running the programs optimized in exactly the same way on a normal direct-mapped cache. This represents an upper bound on the miss rate achievable without excluding instructions. It is possible to approach the performance of the cache with an uncached area with a normal direct-mapped cache by positioning instructions that should be excluded so that they will interfere with each other rather than instructions that should be cached. Column 5 presents performance of the programs without optimization on a normal direct-mapped cache. The miss rate is double that of the optimized rate. The final column shows miss rates for an 8-way set associative cache with 16 word blocks and a single word fetch back. It is identical to the MipsX instruction cache[3] except the MipsX cache fetches back two instructions on a miss. The optimized performance on a direct-mapped cache is substantially better. Note that this is not quite a fair comparison since the direct-mapped cache requires more tags.

| | OPT | optimized some cached | optimized all cached | direct mapped | set assoc. |
|---|---|---|---|---|---|
| bigfm | 7.6 | 10.3 | 11.5 | 20.7 | 19.1 |
| ccal | 15.9 | 18.1 | 22.3 | 31.9 | 32.4 |
| compare | 2.9 | 3.7 | 5.3 | 15.9 | 7.8 |
| dnf | 1.3 | 2.1 | 2.5 | 16.6 | 9.5 |
| hopt | 6.9 | 11.2 | 14.0 | 30.4 | 20.4 |
| macro | 8.5 | 12.8 | 15.2 | 23.4 | 22.0 |
| pasm | 8.7 | 11.0 | 16.8 | 27.0 | 25.7 |
| pcomp | 14.7 | 20.2 | 23.7 | 32.8 | 32.1 |
| simu | 15.2 | 16.8 | 19.5 | 24.8 | 22.8 |
| upas | 13.6 | 18.9 | 21.4 | 27.5 | 28.7 |
| avg. | 9.5 | 12.5 | 15.2 | 25.1 | 22.1 |

**Table 2:** Miss Rates for Various Cache Organizations, Size = 512 Words

Table 3 and Figure 10 show miss rates for optimized with an excluded range, optimized with all instructions cached, and unoptimized direct mapped for a range of cache sizes. We can understand how performance changes by looking at the extremes. For very small caches, an increasing fraction of the misses are essential, ie. are caused by the small cache size irrespective of the replacement policy or how the program is restructured. Thus for small caches the improvement from restructuring decreases. For very large caches, the programs all fit comfortably in the cache and misses result from simply loading the program into the cache. Restructuring does not help. The maximum improvement in miss rate occurs at a cache size of 512 words with an improvement of 12.6% misses. This maximum point depends strongly on the benchmarks chosen. For other programs the maximum point will be different. The highest relative improvement occurs at a cache size of 8K words with a reduction in the miss rate by a factor of over 4 times. While this represents a fairly small improvement of only 1.7% in miss rate, this may still be a significant improvement of overall performance. Machines with large caches often have large miss penalties. For example, if the miss penalty is 10 cycles, the overall performance could increase by up to 17%. The final column in Table 3 shows the percentage increase in program path length caused by the insertion of branches to reposition the code. The increase is small compared to the decrease in instruction cache miss rate.

| size (words) | optimized some cached | optimized all cached | direct mapped | path length increase |
|---|---|---|---|---|
| 128 | 40.1 | 45.9 | 48.8 | 0.6 |
| 256 | 21.5 | 26.2 | 33.7 | 0.5 |
| 512 | 12.5 | 15.2 | 25.1 | 0.4 |
| 1024 | 7.1 | 8.6 | 17.5 | 0.3 |
| 2048 | 3.4 | 4.2 | 9.7 | 0.2 |
| 4096 | 1.2 | 1.3 | 4.8 | 0.1 |
| 8192 | 0.5 | 0.4 | 2.2 | 0.0 |
| 16384 | 0.6 | 0.5 | 0.7 | 0.0 |

**Table 3:** Miss Rates for Direct-Mapped Caches (%)

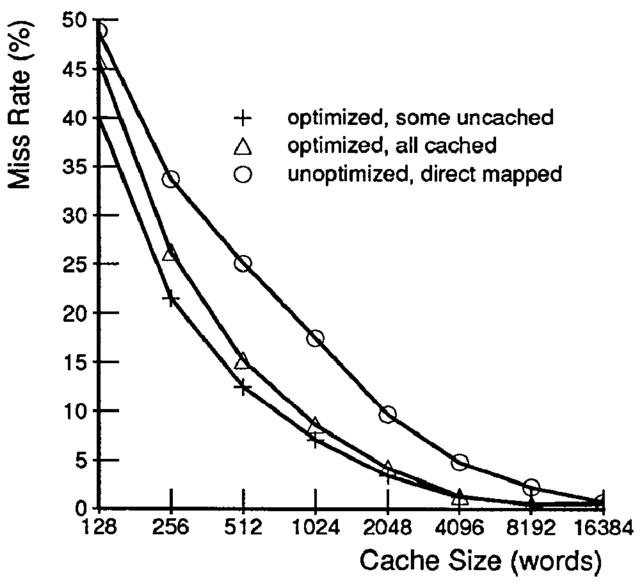


**Figure 10:** Miss Rates for Direct-Mapped Caches

So far we have assumed that the input used to generate the profile file is the same as the input that the program will be run on. The true performance will depend on how much different the program behaves for different inputs. For best results, the program should be run on several inputs representative of the inputs that the program will be used for and the profile counts should be added together. Figure 4 shows the same programs, with the same inputs, on the same cache as Figure 2 but the profile was generated with a different input. Only one input file was used. For example, with the pascal front-end benchmark *upas* the profile file was generated using the Stanford benchmark *eightqueens* and then run on the benchmark *puzzle.* With the exception of *dnf*, the benchmarks do nearly as well as previously. For *dnf* the new input does not use a substantial part of the program used with the original input and this part was excluded from the cache. If we used a large group of profile inputs, this problem would be less likely to occur.

| | some cached | all cached |
|---|---|---|
| bigfm | 10.0 | 11.1 |
| ccal | 21.6 | 25.0 |
| compare | 3.8 | 5.3 |
| dnf | 12.5 | 12.8 |
| hopt | 10.7 | 12.8 |
| macro | 14.1 | 16.7 |
| pasm | 11.4 | 16.2 |
| pcomp | 18.6 | 18.7 |
| simu | 17.7 | 20.8 |
| upas | 23.7 | 23.6 |
| avg. | 14.4 | 16.3 |

**Table 4:** Optimized Miss Rates with Profile from Different Execution

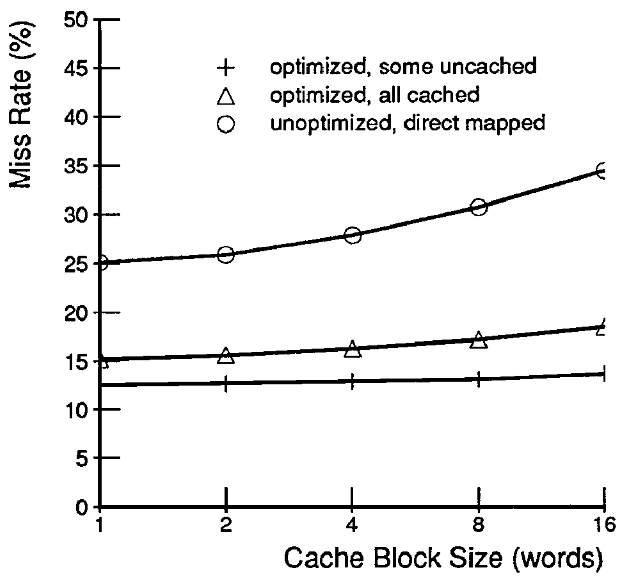


**Figure 11:** Miss Rates for Various Block Sizes, Single Word Fetch

All the data presented above, with the exception of the set-associative cache in Table 2, has been for caches with single word blocks. Figure 11 shows how direct-mapped miss rates, both with and without optimization, are affected by block size. Block size is varied from 1 to 16 words. Sub-block replacement is used, and the fetch-back is kept at a single word. The optimized miss rate increases relatively slowly with increasing block size. While the algorithm presented here does not take block size explicitly into account, by packing as many things into the cache as possible, it tends to pack the blocks as well. Finally, we can now make a direct comparison between the set-associative MipsX cache and an optimized direct-mapped cache. The miss rate of the MipsX cache is 61% or 19% higher than an optimized direct-mapped cache with and without an uncached address range respectively.

## Conclusion

We have presented an algorithm that reduces instruction cache misses for direct-mapped caches. The algorithm has provably good performance for a restricted class of programs. For typical system programs, the algorithm substantially improves the effectiveness of cache. Typically, the optimization is as effective as tripling cache size.

As long as instruction cache miss rates remain a problem, the method presented here can be useful. Currently, processor speeds are increasing faster than memory speeds. While this continues, the instruction cache will remain a bottleneck. While the results presented here are only for instruction caches, the technique may prove useful as well for combined I&D caches.

## References

1. Smith, A. J., ''Cache Memories'', *Computing Surveys*, Vol. 14, No. 3, September, 1982, pp. 473-530.

2. Horowitz, M., Chow, P., et. al., ''MIPS-X: A 20 MIPS Peak, 32-Bit Microprocessor with On-Chip Cache'', *IEEE Journal of Solid-State Circuits*, Vol. SC-22, No. 5, October, 1987, pp. 790-799.

3. Agarwal, A., Chow, P., Horowitz, M., Acken, J., Salz, A., and Hennessy, J., ''On-Chip Instruction Caches for High Performance Processors'', *Proc. of the Conference on Advanced Research in VLSI*, Losleben, P.,ed., Stanford University, Stanford, Ca, March 1987.

4. Hill, M.D., *Aspects of Cache Memory and Instruction Buffer Performance*, PhD dissertation, University of California, Berkeley, November 1987.

5. Przybylski, S., Horowitz, M., Hennessy, J.L., ''Performance Tradeoffs in Cache Design'', *Proc. 15th Sym. on Computer Architecture*, Honolulu, Hawaii, June 1988.

6. Kernighan, B.W., ''Optimal Sequential Partitions of Graphs'', *JACM*, Vol. 18, No. 1, 1971, pp. 34-40.

7. Hatfield, D.J., and Gerald, J., ''Program Restructuring for Virtual Memory'', *IBM Systems J.*, Vol. 10, No. 3, 1971, pp. 168-192.

8. Ferrari, D., ''Improving Locality by Critical Working Sets'', *CACM*, Vol. 17, No. 11, Nov., 1974, pp. 614-620.

9. Ferrari, D., ''The Improvement of Program Behavior'', *Computer*, Vol. 9, No. 11, Nov., 1976, pp. 39-47.

10. Baer, J.L., and Caughey, R., ''Segmentation and Optimization of programs from Cyclic Structure Analysis '', *Proc. AFIPS*, 1972, pp. 23-36.

11. Hartley, S. J., ''Compile-Time Program Restruction in Multiprogrammed Virtual Memory Systems'', *IEEE Transactions on Software Engineering*, Vol. 14, No. 11, November, 1988, pp. 1640-1644.

12. Fabri, J., *Automatic Storage Optimizations*, PhD dissertation, NYU, 1979.

13. Abu-Sufrah, W., ''Identifying Program Localities at the Source Level'', Technical Report UIUCDCS-R-82-1108, Dept. of Computer Science, Univ. of Illinois, October 1982.

14. Thabit, K.O., *Cache Management by the Compiler*, PhD dissertation, Rice University, Nov. 1981.

15. Chow, F., ''Private Communication''.

16. Samples, A. D., and Hilfinger, P. N., ''Code Reorganization for Instruction Caches'', Technical Report UCB/CSD 88/447, University of California, Berkeley, October 1988.

17. Belady, L.A., ''A Study of Replacement Algorithms for a Virtual-Storage Computer '', *IBM Systems J.*, Vol. 5, No. 2, 1966, pp. 78-101.