# Compile Time Instruction Cache Optimizations

Abraham Mendlson[1], Shlomit S. Pinter[1] and Ruth Shtokhamer[2]

[1] Dept. of Electrical Engineering
Technion, Haifa Israel
[2] Dept. of Computer and Information Sciences
University of Delaware, Delaware U.S.A

**Abstract.** This paper presents a new approach for improving performance of instruction cache based systems. The idea is to prevent cache misses caused when different segments of code, which are executed in the same loop, are mapped onto the same cache area. The new approach uses static information only and does not rely on any special hardware mechanisms such as support of non-cachable addresses. The technique can be applied at compile time or as part of object modules optimization. The technique is based on replication of code together with algorithms for code placement. We introduce the notion of abstract caches and present simulation results of the new technique. The results show that in certain cases, the number of cache misses is reduced by two orders of magnitude.

## 1 Introduction

As the speed of a processor increases, the penalties for cache misses become more severe. In order to improve performance, modern computers use a fast clock rate, long pipelines and cache memories, but their access time to the main memory remains relatively slow. Reducing instruction cache misses can improve the overall CPI (clocks per instruction) rate [6] that the processor can achieve. Recently, some widely used benchmarks revealed that there is an important class of programs which show a significant amount of instruction cache misses [8, 9, 2]. Hennessy and Patterson [5] distinguish between three sources for cache misses: (1) *Compulsory* (footprint [13]) – a miss caused when a processor accesses a cache block for the first time. (2) *Capacity* – a miss caused when the cache is too small to hold all the required data and (3) *Conflict* – a miss caused in set associative architecture when the replacement algorithm removes a line and soon fetches it again. According to [8] most of the instruction cache misses are of the conflict type and occur when different code segments within a loop are mapped onto the same cache set.

This paper presents a new approach for reducing instruction cache misses. It uses code replication together with replacement optimization in order to decrease conflict misses, and is based on static information only (information that can be derived at compile time). Conflict cache misses inside loops can be totally eliminated if the loop is small enough to fit in the cache and the code is mapped into the cache without an overlap; such a mapping is achieved by replicating instructions and properly mapping the copies to different sets of the

cache. However, the replication increases the size of the code and the number of compulsory misses. On the other hand, the misses optimization problem based on the placement paradigm alone is an NP-complete problem and in practice it was found to be ineffective in many cases.

Most instruction cache optimization techniques use dynamic information; i.e., profiling information, which is gathered from executing the code on a selected set of input data. The Pixie [3] tools [11] and [10] use profiling information to find better placements for code segments. Information gathered dynamically is also used in [3, 8] for avoiding fetching into the cache instructions either when they are used only once before being purged from the cache, or because they might conflict with other code in the loop. Such instructions are left in the main memory (non-cachable). The effect of inlining procedures on the performance is discussed [9]. In [4], code fragments are repositions so that the cache line will not be polluted (will not contain instructions that are never referenced). All of these techniques were found to be successful mostly for small caches.

Our algorithm works in two phases. In the first one, instructions executed in a loop are arranged relative to each other so that no conflict misses occur if the cache can hold all of them. At this stage, it is assumed that the code is replicated when needed (e.g. a procedure could be replicated if it will be needed in two loops in different relative locations). In the next phase, the expended program (with its replications) is partitioned into parts that fit the cache size (termed abstract caches); during this process, the replications of a code assigned to a single abstract cache are merged. Our experiments show that the total increase in code size is around 5% to 10%. This is followed by heuristics for placing the code of each abstract cache (selecting its address in main memory) and globally trying to merge replications that can each be mapped to the same set in their relative abstract caches.

The new method improves the performance of most small and medium caches as long as the size of inner loops are smaller then the size of the cache. Several of our experiments show an improvement of two orders of magnitude in the number of instruction cache misses. Thus, it seems that the saving of conflict misses when a loop is executed outperforms the addition of compulsory misses due to the extra copies. Note that the new cache optimization can be applied at compile time or as part of object modules optimization (for example in the OM system [12]).

The rest of this paper is organized as follows: in Section 2, we provide a rigorous framework for instruction cache optimization algorithms by formulating the general problem and the optimization criteria. The first algorithm that partitions the flow graph into cache sized program parts (abstract caches) is presented in Section 3. In Section 4, we describe how program segments of different abstract caches are placed in the program address space. Performance benchmark results are presented in Section 5 and conclusions in Section 6.

[3] Pixie is a trademark of MIPS Computer Systems, Inc.

## 2 Definitions and the Cache Mapping Problem

In this section, we define the general setting of the problem.

### 2.1 The Cache

For simplicity, we consider in our framework a direct mapped instruction cache. The cache memory is partitioned into *lines* (sometimes called *blocks*) that can each hold a few memory words. The entire block is fetched or replaced in any cache operation that involves the memory. When using a direct mapped cache, any physical address has exactly a single cache line that can hold it. We discuss later on the adaptation of our method to different cache organizations as well.

### 2.2 The Cache Mapping Problem

The number of misses caused by a program depends on the input data, but since the placement of code segments in the virtual address space is fixed for all possible inputs, it may happen that a code mapping may be optimal for one execution and non-optimal for another. Thus, the mapping problem is defined with respect to a *typical set of executions* determined by some set of input data. The question of what is a good typical execution set depends on what information is available and whether we are looking to optimize the most frequently used execution, an average occurring execution or the worst possible case (i.e. choose the mapping for which the worst execution is the best). Note that many different executions of a single mapping can still be optimal. For example, whenever a conditional branch (which does not close a loop) occurs, the code segments on the true branch can be mapped in such a way that they will fall on the same cache lines as the code on the false branch owing to the fact that they never execute together.

**The Cache Mapping Problem:** *Given a set of cache parameters, a program, and a typical execution set, find a placement of the program parts (in the memory address space) that minimizes the total number of misses for the typical execution set.*

### 2.3 Program Structures and Intervals

To simplify our analysis of the program behavior, the input programs are assumed to be well-structured. Thus, loops are properly nested and every loop has a single exit point; a procedure call is always returned to the instruction immediately following the call (see [1] for justifications of the assumption). We present such a program by the Nested Flow Graph (NFG), which is a flow graph [1] augmented with information about the nesting structures of loops and the calls for procedures.

In the flow graph, every node represents a basic block, and an edge from node $v$ to node $u$ indicates that the execution of $u$ must follow that of $v$. In the NFG,

there is another type of node called a *level node*; such a node represents a loop (its level) or a procedure call (when its body is not inline). In addition, we use *nesting edges* to indicate a change in the nesting level. The NFG is generated from the flow graph by replacing each loop closing back edge with a level node and directing such an edge to point into that level node (see Figure 1). The flow edges that enter the loop are now pointing to the level node and the flow edge leaving the loop is now drawn from the loop level node. Similar construction is applied to procedure calls. A nesting edge is drawn from a level node to the node that represents the beginning of the loop's (or procedure) body. For recursive (direct or indirect) procedure calls, we stop generating level nodes when there is a path from a previous call to the current call and the path contains a nesting edge.

The code fragment in Figure 1 has three loops and three procedure calls. Note that procedure B appears twice in the NFG, once when it is called from `loop2` and the second time when it is called from `loop3`.

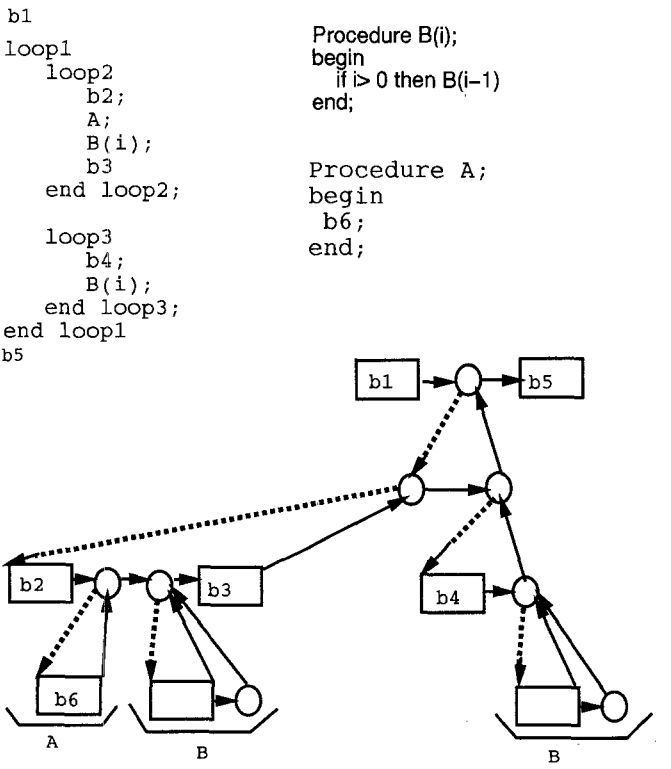


**Fig. 1.** A code fragment and its NFG

Our two basic concepts – intervals and nesting interval trees – can now be defined based on the NFG of a program.

**Definition 1** *An* `interval` *is a maximal connected sub-graph of the NFG that may not include parts nested under its level nodes. A* **segment** *is a connected sub-graph of an interval.*

Blocks `b1` and `b5` together with the level node between them are an example for an interval.

**Definition 2** *A* **Nesting Interval Tree (NIT)** *of a nested flow graph is an ordered tree (order on the edges leaving a node) whose nodes are the intervals of the NFG and its edges are the nesting edges connecting the intervals.*

The nesting interval tree of the NFG in Figure 1 is presented in Figure 2.

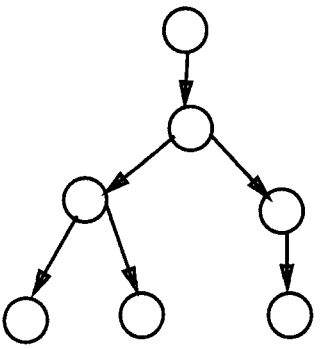

**Fig. 2.** The NIT of the code fragment in Figure 1

**Definition 3** *Two intervals that represent the same expanded code (e.g. copies of a procedure's body replacing its calls) are called* **matching intervals** *(similar definition is used for segments).*

Note that matching intervals may appear in different places of the NFG. In particular, the body of a procedure which is called from different intervals is presented by different embodiments.

**Definition 4** *The* **locality tree** *of a node, v, of the NIT is the sub-tree derived from the NIT by taking all the nodes reachable from v (on directed paths).*

Due to the redundancy (replication) of a code in the NFG a locality tree with a root $v$ contains all the basic blocks (within the intervals nodes) for executing the program fragment represented by $v$. This redundancy will be used later as a powerful optimization technique.

Every mapping algorithm must place all the intervals (their basic blocks) that cover the program in the program address space so that: (1) intervals (other than matching intervals) do not overlap in memory; (2) matching intervals (e.g. those that were generated due to procedure calls) can be mapped onto the same address or onto different addresses (their names must then be modified).

The basic elements mapped by our algorithms are the intervals and segments of the program's NFG. Segments of an interval are used only when it is too large to be mapped entirely onto the cache. If the system can bypass the cache, it can be assumed that all the basic blocks that are not part of any loop are marked as non-cachable and are not part of any interval.

# 3 Partitioning the NFG Into Abstract Caches

The optimization algorithms use the NFG representation and the locality trees in order to map the program onto the main memory. The replications of code in the NFG are now used for generating a map that fits loop bodies (presented by the locality trees) within a single cache as long as possible. This local optimization uses abstract caches which are next presented together with some of their properties. In Section 3.2, we present a global optimization for mapping the instructions of the abstract caches into memory.

## 3.1 Abstract Caches

**Definition 5** *An* `abstract cache` *is a connected subgraph of the NFG whose size (the total space taken by its basic blocks when its matching intervals are counted only once) is not greater than the size of the instruction cache divided by its associativity. A partition of a graph into abstract caches is an* `abstract cache cover (or cover)` *of the graph.*

Two important properties of abstract cache covers are summarized in the following lemmas:

**Lemma 1.** *Let C be an abstract cache cover of a nested flow graph G such that no two matching intervals are in different abstract caches. Then, all the basic blocks of G can be mapped onto a cache, whose size is at most the sum of all the abstract caches of C, without causing any conflict miss.*

Proof: In each of the abstract caches, all the basic blocks can be mapped sequentially. The total sum of the caches will be large enough to keep each program's line in a different cache line. Therefore, only footprint misses can occur.

**Definition 6** *Two abstract caches can be* `mapped independently` *onto memory if their joint cover has no matching intervals.*

**Lemma 2.** *Given a program NFG with an abstract cache cover. If no loop is partitioned among different abstract caches and every abstract cache can be mapped independently of other abstract caches, then only compulsory cache misses can occur.*

Proof: Since no loop is partitioned between abstract caches, the program at the top level can be viewed as a sequence of abstract caches (phases of the program execution). As soon as the execution inside an abstract cache is terminated, the program will never access that code again (since the caches can be mapped independently). Since no abstract cache is larger than the physical cache, with Lemma 1, only compulsory cache misses can occur.

When replication is used, matching intervals of different abstract caches become distinct by a proper renaming. Thus, the number of footprint misses may still increase. To further reduce the additional footprint misses, global minimization of replicated code is needed. Such an optimization is employed during the final placement.

### 3.2 An Algorithm for Creating an Abstract Cache Cover

Since optimal mapping is defined with respect to a typical set of executions, we next discuss the set assumed by our algorithms. As we do not use profiling or run-time information, only the program structure can be used in our algorithms. We assume that the branches of a conditional are equally taken and the number of iterations in all loops are the same; thus the instructions in a loop which is nested within another loop will be executed many times more as the number of iterations per loop. Lastly, we assume that when the first instruction of a loop or a procedure is accessed its entire interval will be accessed next. In the discussion section, we consider a different set of assumptions and their effect on the mapping.

Selecting a cover for the NFG is done by traversing the nested interval tree starting from the leaves; in this way we try to keep locality trees in the same abstract cache as long as they fit. Each leaf interval is assigned to a different abstract cache. When traversing up the levels of the tree, abstract caches are merged as long as their total size (matching intervals are considered only once) is less than the cache size.

**Abstract Cache Partitioning Algorithm**

- Input: The NFG representation of the program, its nested interval tree $T = (V, E)$ of height $H$, and a cache size C.
- Output: A partition (cache cover) of the intervals into abstract caches each of size not greater than C.
- Step-1
  Assign each leaf interval of the NIT to a different abstract cache. If the size of an interval is larger than C, repeatedly extract from the interval a set of continuous basic blocks (segments) of size no greater than $C$ and assign it to a new abstract cache.
- Step-2
  `for i=` $H$`-1 down to 0 do`
  * for every level node in level (height) `i`, make a cache cover list from the abstract caches of its descendent nodes; during this process merge

abstract caches whenever possible and leave only a single copy when several matching intervals are assigned to the same abstract cache.

* merge the cover lists of level `i` in a way similar to the above

- Step-3
  The cache cover list of the root is the cache cover of the program

Note that similar (matching) basic blocks, like those of called procedures, can appear in more than one abstract cache but will appear at most once in each abstract cache. Abstract caches preserve the locality of memory references generated by the processor since the graph is partitioned along the nesting levels of the NFG.

# 4 From Abstract Caches to the Process Address Space

In the last phase of our optimization the blocks in each abstract cache are positioned in the program address space (global optimization). The placement algorithm is based on *the basic placement algorithm* which is valid for the cases described in Lemma 2: In Section 4.2, we extend it to handle the general case. The extended algorithm applies heuristic methods for the global optimization.

## 4.1 The Basic Placement Algorithm

The basic placement algorithm assumes that no loop is divided among different abstract caches, and matching program intervals that reside in different abstract caches are renamed.

- Input: A list of abstract caches (ACs) covering the NFG.
- Output: A placement (address) for locating the intervals in the memory space.
- Data structures:
  `Unplaced_list` : A list of all ACs which have not been placed.
  `Placed_list` : A list of all ACs which have already been placed.
  `Active_AC` : The AC currently being placed.
  `Program_address_space` : An image of the program address space. There is an indication of each location whether it has already been placed or not.
  `LPC` : Points to the highest address in the program address space that has been used.
  `Temp_cache` : An array whose size is the same as the system cache. Used for placing the intervals of each AC.
- Initialization conditions: All the ACs are in the `Unplaced_list`, and `LPC` is 0.
- `while Unplaced_list` is not empty `do`
  1. Choose an AC from `Unplaced_list` to be the `Active_AC`.
  2. Place all intervals (their basic blocks) of `Active_AC` in `Temp_cache` continuously. This can always be done under the current assumptions (Lemma 2).

3. Copy `Temp_cache` to the program address space, starting at address `LPC`+1, and update `LPC`.
4. Put the `Active_AC` in `Placed_list`.

For mapping in the general case, there are two aspects to consider. The first one is when a replication is worthy (which affects the AC cover) and the second is the order in which to select the `Active_AC` in Step 1.

### 4.2 Extended Placement Algorithm — a Heuristic Approach

Since only static information is used, we assume that the entire cache is flushed when the processor exits an abstract cache. The basic algorithm is extended by checking how the placement of a replicated interval in one abstract cache can affect its placement when it appears in another abstract cache.

Consider the two calls in the inner loops of Figure 3. If the two copies of `B` are located in the same place relative to their abstract caches then no duplication is needed; otherwise extra footprint misses are generated in each iteration of `loop1`. Note that in both cases no conflict misses are generated for the inner loops. The exact number of misses depends on the number of iterations and the size of `B`. In the static case, where no information is provided on the number of times each loop is executed, we observe that preventing misses at innermost nesting loops is highly desirable.

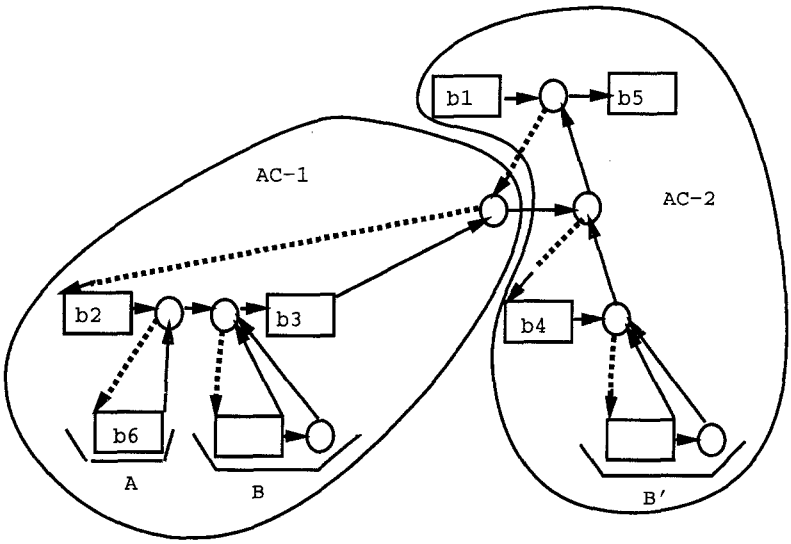


**Fig. 3.** Partition to abstract caches

Our extended placement algorithm is based on the basic algorithm. Its input is an abstract cache cover and the NFG in which matching intervals were not yet renamed. It uses an extra data structure `Placed_interval` to hold all the intervals which have already been placed in the program address space. Initially, `Placed_interval` is empty.

In Step 1, the policy for choosing the `Active_AC` is changed to follow a heuristic from the set H1 that is described later.

Step 2 becomes:

1. Try to place in `Temp_cache` all the intervals that belong to the `Active_AC` and which their replications appear in the `Placed_interval` list. The place in `Temp_cache` should coincide with the mapping onto the cache of the copy (one of them). If an interval cannot be placed in such a way, use a duplication policy from the set H2 which we describe later.
2. Place all the remaining intervals of the `Active_AC` in `Temp_cache`. If an interval cannot be placed, (not enough space), then it will be placed with the next abstract cache.

In Step 3, we must add the update of the `Placed_interval` list. Step 4 remains as in the basic algorithm.

In the algorithm, two sets of heuristics are used. The first, H1, determines the order in which abstract caches are chosen to be placed, and the second, H2, is used for deciding whether or not to duplicate an interval.

The order in which abstract caches are selected can be one of the following: *deep first* — choose the innermost interval to be the `Active_AC`, or a *sequential* selection which chooses some order of abstract caches which the program is most likely to visit, or lastly, a *random* choice can be made.

Each time an abstract cache is chosen, all of its intervals are mapped onto the process address space. But an interval that has a copy which is already placed in memory sometimes cannot be put in the temporary cache in the place to which its copy is mapped (the place is taken). In such a case, three policies for replications (H2) are suggested. The first is to *always replicate* — take a copy of the interval and rename it. The new interval is free of all other constraints and so can be mapped. The second is *never replicate* — use only a single version of the interval (the first one that was mapped). The last one is *sometimes replicate* — if a fraction of the desired space is occupied (overlapped), then a threshold limit is used to decide upon replication.

## 5 Simulation Results

This section presents simulation results of the UNZIP program which is used to uncompress and open '.ZIP' programs. A comparison is made between the program running with our optimization technique and without it. In our implementation, only procedures were replicated whenever the corresponding place in the cache was occupied by other parts of the blocks of the active abstract cache. The abstract caches were selected using a preorder search of the NFG.

First we describe the methodology being used for generating the traces and then we present and discuss the results.

### 5.1 Generating the Traces

We use the AE tool [7] to relocate and replicate basic blocks of programs in the main memory. This enables us to implement our algorithm independently of the compiler being used.

AE is a tool used for generating traces. It has three major phases: (1) generating static information about the program. This includes the control flow graph, a list of basic blocks with their respected addresses in memory, procedure calls, nesting of loops etc.; (2) gathering dynamic information on a particular run (using some input); (3) generating the trace by combining the program source file with the information from 1,2.

We modify the addresses of basic blocks generated in phase 1 of the AE tool in order to relocate and duplicate them according to our technique. The relocation is implemented by changing the starting point of a basic block, while replication is implemented by duplicating the description of the basic block and changing its ID and starting point. Note that by using this technique, the program keeps the same flow of control, but the addresses being generated may be shifted.

The flexibility of the AE tool is well suited to our purposes. On the other hand, the AE does not generate traces for library routines so we could not present the full capability of our technique. The instruction traces being generated were fed into the Dinero cache simulator [5].

### 5.2 The Simulation Results

The UNZIP program was chosen as an example for a medium size program. Its object code is 82K-bytes long, it contains 32 procedures and 22 untraced library procedures, 41 loops and 580 basic blocks. All the traces we used were 5.4M instructions long.

The cache performance is measured in terms of miss ratio for different cache and block sizes. Similar to other relevant works, we choose the cache organization to be direct mapped.

Table 1 presents the results for a cache size of 512 bytes. For such a small cache, our algorithm improves the cache miss ratio from 6% for a block size of 32 bytes to 12% for a block size of 4 bytes. The results indicate that the abstract caches were too small to hold significant loops, e.g. loops containing procedure calls. Note that a loop which does not call any procedure can be assumed to be sequentially placed for both cases.

For a cache size of 1024 bytes, our algorithm reduces the cache miss ratio compared to the C compiler by 30-35% depending on block size (see Table 2).

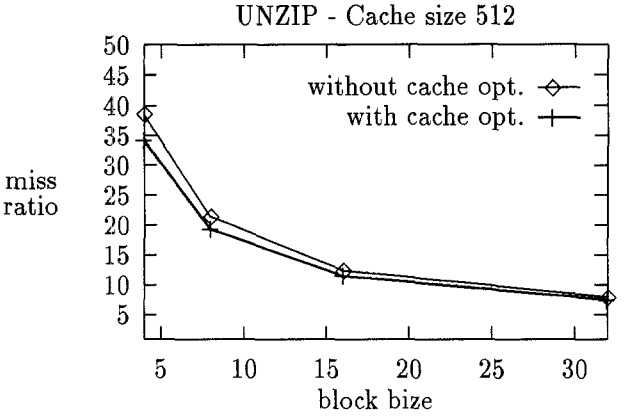


**Table 1.** UNZIP: Cache size 512 - 5,400,000 instructions

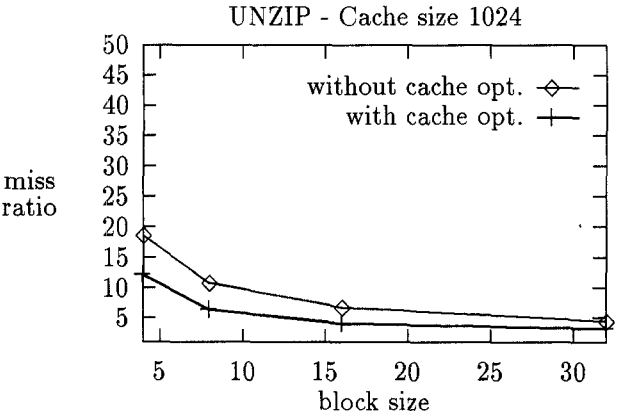


**Table 2.** UNZIP: Cache size 1024 - 5,400,000 instructions

Improvement of a few orders of magnitude in the cache misses is achieved with a cache size of 2048 bytes. Table 3 presents the miss ratio for this case. Due to the difference in magnitude, we present the graph in logarithmic scale.

The improvement presented in Table 3 is explained by the behavior of the program execution. Logically, the UNZIP program runs in phases. Each phase is typically composed of nested loops with procedure calls. Since the cache was large enough, our replication and relocation technique managed to place the called procedures of each phase in a conflict free manner. Thus, the total amount of misses is relatively close to the footprint of the execution and is not dependent

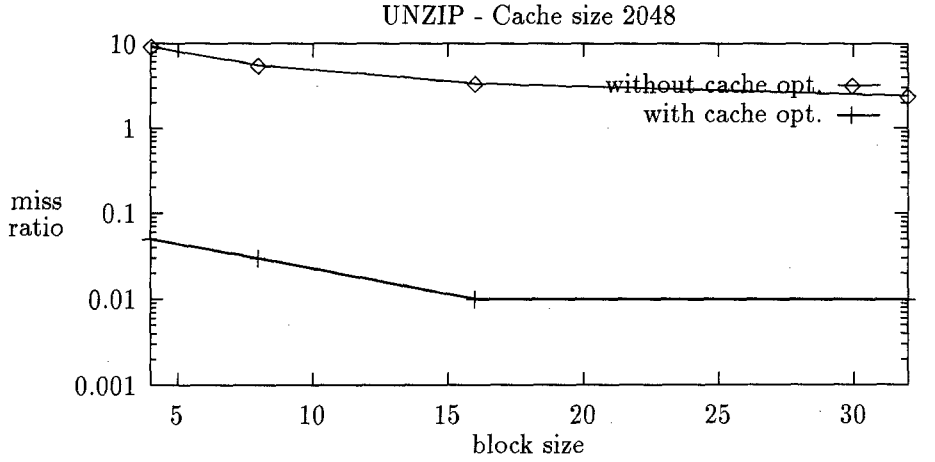


**Table 3.** UNZIP: Cache size 2048 - 5,400,000 instructions

on the number of times that the loops are executed.

Without replications, it is not possible to eliminate the conflicts caused by procedures called in different phases. In such a case, as soon as a cache conflict occurs in a loop, the number of cache misses depends on the dynamic nature of the program, i.e. how many times the loop is executed.

To support these conclusions we compared the footprint of our run (using a huge cache size) against the actual number of cache misses (using a cache size of 2048 bytes) with and without the cache optimizations. The results for block size of 32 bytes are 242 cache misses (footprint) compared with 474 cache misses using the cache optimizations and 130095 cache misses in the original placement generated by the compiler.

It is very likely that the above results would improve with a pre-fetch mechanism, since the replicated code is executed sequentially.

## 6 Conclusions and Future Work

We presented a new approach for optimizing instruction cache systems that can be applied during both compilation time and object modules optimization. The main idea in this approach is to prevent segments of codes that are executed in the same loop to be mapped onto the same cache area. Our approach differs from other techniques that use only compile time information and is based on two basic techniques: duplication of code and sophisticated placement algorithms.

We integrated our algorithm into a version of the AE tool which is based on the gcc compiler. The use of indirect methodology to examine our algorithm

gives us a great deal of flexibility and the ability to get meaningful results in a reasonable amount of time. However, this approach has some limitations. In particular, it limits us to use relatively small input programs. Since a small program has a small footprint, the efficiency of our approach can be exemplified for small and medium caches only. We believe that our methodology can be even more effective when applied to larger programs.

For simplicity, we choose to use direct mapped cache memories. Nevertheless, the same algorithm can be easily extended to other cache organizations. If cache associative organization is used, the size of the abstract cache is reduced to the size of the cache divided by the associativity of the cache. Thus, more intervals can be mapped onto the same cache set before a miss occurs.

The algorithm presented here can be improved by further taking the program structure into account when placing the basic blocks. For example, the different paths of an `if then else` structure can be mapped onto the same cache area so that the cache will be used more efficiently to benefit smaller caches. Another possible extension can consider non-well structured programs as input.

The compilation technique presented in this paper was found to be effective for small and medium cache size and for inner loops of size smaller than the cache size. Note that we can use this information to optimize the amount of loop unrolling; i.e., the total size of the unrolled loop should not exceed the size of the cache.

Currently, we are investigating new improvements that include: the effect of different architecture mechanisms such as branch prediction, and instruction prefetching. The effect of different cache organization and parameters, and the effect of different application structures on the overall performance of our method.

## References


1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.
2. D. Bernstein, D. Cohen, Y. Lavon, and V. Rainish. Performance evaluation of instruction scheduling on the IBM RISC system/6000. In *25th Annual International Symposium on Microarchitecture*, pages 226–235, Portland, Oregon, December 1992.
3. C. Chi-Hung and H. Dietz. Improving cache performance by selective cache bypass. *Hawaii International Conference on System Science*, pages 277–285, 1989.
4. R. Gupta and C. Chi-Hung. Improving instruction cache behavior by reducing cache pollution. In *Proc. of Supercomputing '90*, pages 82–91, New-York, November 1990.
5. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Kaufman, 1990.
6. W. W. Hwu and P. H. Chang. Achieving high instruction cache performance with an optimizing compiler. In *The* $16^{th}$ *International Symposium on Computer Architecture*, pages 242 – 251, Jerusalem Israel, May 1989.
7. J. R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software Practice & Experience*, 20(12):1241–1258, 1990.

8. S. McFarling. On optimizations for instruction caches. In *The $3^{rd}$ International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–192, 1989.
9. S. McFarling. Cache replacement with dynamic exclusion. In *The 19th Annual International Symposium on Computer Architecture*, pages 191 – 200, Gold Coast, Australia, May 1992.
10. K. Pettis and R. C. Hansen. Profile guided code positioning. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 16 – 27, 1990.
11. M. D. Smith. Tracing with pixie. CSL-TR-91-497 91-497, Stanford University, Stanford, CA 94305-4055, November 1991.
12. A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, 1993.
13. D. Thiébaut and H. S. Stone. Footprints in the cache. *ACM Transaction on Computer Systems*, 5(4):305–329, November 1987.