Local Search For Optimizing Instruction Cache

by

Amit Khetan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999

© Amit Khetan, MCMXCIX. All rights reserved.

The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis document
in whole or in part.

Author

Department of Electrical Engineering and Computer Science

May 21, 1999

Certified by

David R. Karger
Associate Professor
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Students
Local Search For Optimizing Instruction Cache

by

Amit Khetan

Submitted to the Department of Electrical Engineering and Computer Science on May 21, 1999, in partial fulfillment of the requirements for the degree of Masters of Engineering in Computer Science and Engineering

Abstract

The following study investigates the use of compile time code reordering algorithms to reduce cache miss rates. It is found that code reordering with direct mapped caches can perform nearly as well as fully associative caches in most cases. Heuristics for code layout are examined and modifications are proposed resulting in a metric which accurately measures potential cache conflict. Using the heuristic as a model, relations to several well known combinatorial optimization problems are explored. Up until now the most successful algorithms for code layout have all been “greedy” algorithms. Here, a new framework for finding good code placements is developed. The new approach uses “local search” techniques instead of greedy placement. A comparison of the performance of local search placement against current techniques shows that the algorithm performs comparable to the current state-of-the-art. Finally, we compare the cache miss rates with fully-associative LRU and an optimal fully-associative cache to demonstrate the relative importances of good mapping and/or replacement techniques and associative caching.

Thesis Supervisor: David R. Karger
Title: Associate Professor
Acknowledgments

I would like to thank Nick Gloy for his help in providing me with source code and benchmarks, Mike Smith for helpful ideas and criticisms, and my advisor David Karger for keeping me on the right track.
# Contents

1 Introduction 9
  1.1 Thesis Outline .................................................. 10

2 Code Layout Background 12
  2.1 Effects of Code Layout on Instruction Caches .................. 12
    2.1.1 Cache Structure ............................................ 12
    2.1.2 Memory Mapping and Code Layout ............................ 13
  2.2 Code Layout Algorithms .......................................... 14
    2.2.1 Determining Cost Metric .................................... 14
    2.2.2 Optimizing the Cost Metric ................................. 15
  2.3 Benchmarks ...................................................... 15

3 Conflict Metrics and Heuristics 17
  3.1 Conflict Metrics .................................................. 17
    3.1.1 Granularity ................................................ 18
  3.2 Call Graph Based Metric .......................................... 20
  3.3 GBSC metric ...................................................... 22
    3.3.1 Basic GBSC Model ........................................... 22
  3.4 Refinements to the GBSC Model ................................... 23
    3.4.1 Window Size ................................................ 24
    3.4.2 Probabilistic Metric ....................................... 25
    3.4.3 Other Metric Refinements ................................... 27
  3.5 Improving the Metric through Code Transformation ............. 28
3.5.1 Separating Fluff ................................................. 28
3.5.2 Code Linearization ............................................. 30

4 Local Search Algorithms ............................................ 31
4.1 Granularity of Placement ......................................... 31
4.2 Local Optimization ................................................ 33
  4.2.1 Layout and Move Representations .......................... 33
  4.2.2 Computing the Cost Function ............................... 33
  4.2.3 Local Optimization Algorithm ............................ 35
  4.2.4 Drawbacks .................................................. 37
4.3 Simulated Annealing .............................................. 37
  4.3.1 Scheme Outline and Notation .............................. 38
  4.3.2 Single Temperature Run ................................... 38
  4.3.3 Temperature Control ....................................... 39
  4.3.4 Simulation Length vs. Quality Tradeoff .................. 42
  4.3.5 Complexity Analysis ..................................... 42
  4.3.6 Intra-Block Optimization ................................. 44
4.4 Extension to L2 Cache ........................................... 45
  4.4.1 Conflict Metric ............................................ 45
  4.4.2 Algorithm .................................................. 45
4.5 Other Optimizations .............................................. 46
4.6 Creating the Linear Order ....................................... 46
4.7 Results .......................................................... 47
4.8 Comparisons to Associative Caches ............................ 47
  4.8.1 LRU cache .................................................. 48
  4.8.2 Belady’s algorithm ........................................ 49
  4.8.3 Summary of Findings ...................................... 51
  4.8.4 Hardness Results for Direct Mapped Code Layout ...... 51
4.9 Summary .......................................................... 53
4.10 Future Work ..................................................... 53
5 Reductions to Combinatorial problems

5.1 Quadratic Assignment problem
5.1.1 Formulation
5.1.2 Hardness
5.1.3 Algorithms
5.1.4 Lower Bounds
5.1.5 Summary

5.2 MAX k-CUT problem
5.2.1 Formulation
5.2.2 Hardness
5.2.3 Algorithms
5.2.4 Summary

5.3 Related Problems
5.3.1 Traveling Salesman Problem

5.4 Conclusion

6 Conclusion
6.1 Contributions of This Thesis
6.2 Future Work

A Graphs
List of Figures

3-1 Call Graph Based Metric .............................................. 21
3-2 GBSC Metric ............................................................. 23
3-3 Effects of Window Size on the L1 Cache Miss Rate .............. 24
3-4 Effects of Alpha on the L1 and L2 Miss Rates .................... 26

4-1 Pseudocode for Cost Computation ................................. 34
4-2 Simplified Cost Computation for Chunks ......................... 35
4-3 Pseudocode for Local Optimization ............................... 36
4-4 Simulated Annealing at One Temperature ....................... 40
4-5 Effects of Varying Temperature Ratio on Simulated Annealing at Numerous Search Length ......................... 54
4-6 Effects of Simulation Length on Simulated Annealing ......... 55
4-7 Comparison of LRU to Belady’s OPT Cache ..................... 56

A-1 Comparisons of Caching Algorithms ............................... 74
List of Tables

2.1 Benchmark Programs ..................................................... 16

3.1 Effects of Fluff Separation on Primary Cache Miss Rate .......... 29

4.1 Simulated Annealing Quantities and Parameters .................. 39

4.2 Comparison of Local Optimization and Simulated Annealing ..... 43

4.3 Running Times of Code Placement Algorithms ..................... 44

4.4 Comparison of Code Layout Algorithms .............................. 48

4.5 Comparison of SA and LRU ............................................. 49

4.6 Comparison of SA and Belady's Algorithm ......................... 50
Chapter 1

Introduction

Improving the amount of time spent in memory accesses is becoming an increasingly important concern in optimizing overall machine performance. Processor speeds continue to increase much faster than memory speeds, requiring the use of clever mechanisms to avoid the "memory bottleneck." This is especially relevant for instruction memory, since most modern pipelined machines are highly sensitive to delays in the instruction fetch stage. Memory access time is inversely proportional to the size of the memory. Therefore, a small, fast memory cache is an absolute necessity for good performance. The cache prefetches a set of instructions that are likely to be referenced in the near future. Caches work because programs typically reference only a small set of instructions, the working set, at any given time. Ideally a cache should always hold the current working set of code [1]. If the working set is larger than the cache size, than some misses to memory are unavoidable and can only be reduced by increasing the cache size. A more serious concern is if two instructions in the same working set map to the same location in the cache. This results in conflict misses. Reducing conflict misses is a major concern in cache design.

The simplest cache design is a direct mapped cache. Each memory location is mapped to a fixed cache location determined by the low order bits of the address. direct mapped caches are fast in hardware but can result in many conflict misses if multiple "hot spots" in code compete for the same cache lines [2]. The standard tradeoff is to use associative caches, which reduce the number of conflict misses by
allowing a code block to be in any of a set of possible cache locations. However
associative caches have slower hit times than direct mapped caches due to the time
it takes to find the correct code block among its possible locations. A more subtle
drawback is that associative caches make it more difficult to perform speculative
execution, since no data is available until after the hit/miss decision is made. For
these reasons, many modern processors such as the Intel x86 series and the Alpha
21164 use only direct mapped primary caches. An alternative to associativity is to
concentrate on the layout of programs in the memory space. The idea is to use direct
mapped caches but have the compiler reposition the code in order to find a good
mapping of program locations to cache lines.

The default code layout produced by most compilers simply places procedures in
the order that they appear in the source code and preserves the order of object files
from the linker command line. This has a number of obvious drawbacks. First of
all cache misses are left to chance. There is a cache conflict whenever the execution
alternates between two code blocks mapping to the same cache line. However, the
default mapping of procedures to cache lines is almost arbitrary. Even worse, any
optimization to the code that changes the order of the procedure blocks can have
unexpected consequences for the cache miss rate as demonstrated by several studies
[3, 4]. Therefore, the performance of the program may change not just because of the
desired optimization but also due to a much harder to predict change in the miss rate
for the instruction cache. Hence, an intelligent code reordering technique can not only
improve cache performance but also increase the effectiveness of other compile-time
optimizations.

1.1 Thesis Outline

The following study investigates the use of compile time code reordering algorithms
to reduce cache miss rates. We find that code reordering with direct mapped caches
can perform nearly as well as naive fully associative caches without knowledge of the
future. We show how to best capture run-time information for use in code placement
and develop a fast algorithm which generates layouts as good or better than the current state of the art in code reordering.

Chapter 2 describes the basic elements of compiler code placement. First, we present the memory model and the mechanism by which compile time reordering affect the cache miss rate. Next, we describe the two major components of a code layout algorithm: extracting a conflict measure from information about previous runs, and determining the ordering of blocks in the program so as to minimize that measure. Previous approaches in each of these areas are discussed briefly. Finally we provide a description of the benchmarks used to evaluate performance in the later chapters.

Chapter 3 provides a detailed discussion of the heuristic conflict measure used to predict cache conflicts. We rely heavily on the work of Gloy, Blackwell, Smith, and Calder [5, 6] and provide some additional improvements of our own.

Chapter 4 describes the new code block placement algorithm. Up until now the most successful algorithms for code layout have all been “greedy” algorithms. The new approach uses local search techniques to optimize block placement. A candidate code placement is successively improved via local changes until an optimum is reached. The new algorithm is shown to be asymptotically faster than the current greedy approach and gives results that are just as good for one level and two level cache systems. The algorithms are also compared against fully associative caches using LRU replacement and the “optimal” replacement algorithm described by Belady.

The code layout problem is part of a class of problems called “combinatorial optimization problems.” In chapter 5 we model the code layout problems by several well studied problems of this type in the hopes of finding good placement algorithms and tight lower bounds for current and future algorithms.

Chapter 6 summarizes the results of the first five chapters and briefly investigates future research directions. Appendix A contains some graphs of our testing results.
Chapter 2

Code Layout Background

This section provides background material on caches and code layout. We describe the structure of a direct mapped cache system and the method by which code layout can improve the miss rate. We then describe the two basic components of code layout algorithms that will be studied in depth in later chapters: computing a conflict metric and finding a placement to minimize this metric. We conclude with a brief description of the benchmarks used for testing purposes.

2.1 Effects of Code Layout on Instruction Caches

We give a brief review of cache architectures and memory mapping and illustrate the effect of code reordering.

2.1.1 Cache Structure

A memory system is composed of the main memory along with one or more levels of cache. A cache is a small, fast memory unit that stores program instructions or data that have been previously fetched from main memory. We will be dealing with instruction caches that store only program instructions and not program data. Most modern processors have separate instruction and data caches at the primary level although the secondary cache is often unified. A cache is divided up into equal
size blocks called *cache lines*. For example, the Intel Pentium Pro processor has an 8kb primary (L1) instruction cache consisting of 256 lines of 32 bytes each. Most processors also have a secondary (L2) cache that is larger, 256Kb or 512Kb for the Pentium Pro, and often off-chip. Some processors, such as the DEC Alpha 21064 support L3 caches.

Each cache line can store the contents of an aligned block of memory, i.e. the starting address is a multiple of the cache line size. When the CPU tries to fetch an instruction it first looks in the primary cache and then the secondary cache, until finally the instruction is fetched from main memory. The processor computes the address aligned memory block containing the desired instruction and checks if this block is in the cache. If the block is not in cache, the entire block is fetched from the next level in the memory hierarchy, and some line in the current cache has its contents replaced.

### 2.1.2 Memory Mapping and Code Layout

A *direct mapped* cache has a fixed mapping from memory addresses to cache lines, so that a given memory block can only be mapped to a single cache line. For a given memory address the aligned block is computed by looking at the highest order bits, and the corresponding cache line is the low order bits of this block. Other types of caches are *set associative* in which a memory block can be mapped to one of a set (typically 2 or 4) of cache lines, and *fully associative* in which each memory address can be in any cache line. Direct mapped caches are naturally faster and preferred in many modern high-speed processors such as the Intel x86 series and the Alpha 21164.

A *cache mapping conflict* occurs when two instructions have program addresses separated by a multiple of the cache size. Only one of the two instructions can be present in the cache at a given time. A *conflict miss* occurs when one instruction is referenced while the other is still in cache. A code placement algorithm attempts to minimize the number of conflict misses by mapping instructions to specific cache lines. Code placement algorithms can be applied at various levels of granularity. Most techniques focus on the placement of either whole procedures, or of basic blocks

13
which are delimited by branch instructions. The compiler can force blocks to map to specific cache lines by manipulating the order of code blocks in the executable, and if necessary leaving gaps between blocks, to impose an alignment.

2.2 Code Layout Algorithms

A trace based code layout algorithm uses previous executions of the programs to find a good mapping of blocks to cache lines. Other approaches use program analysis to find mappings based just on the structure of the code [7]. Non trace based algorithms, such as the one developed my MacFarling [7], do not have information about the actual sequence of instructions that will be executed and so typically optimize for the worst case execution trace. Trace based algorithms yield better placements since they can exploit information about the actual common instruction sequence.

All trace based code layout algorithms have two parts. The first part involves processing the traces from previous runs of the program to determine the approximate cost in cache misses of any layout. The second part is an algorithm to determine a layout that minimizes this cost.

2.2.1 Determining Cost Metric

The first step in a code placement algorithm is to develop a conflict cost metric which is a function of the possible code layouts and estimates the number of cache misses that the layout will cause. The idea is to exploit information about the order in which instructions are run in previous executions in order to predict the behavior in future executions. In order for this method to be successful, these “typical runs” need to be well chosen and representative of the actual interaction between code blocks that will be encountered during the lifetime of the program. Finding good testing traces can be a difficult problem but since the focus of this thesis is on mapping algorithms, we do not address it here.

The process of extracting information from an execution of a program is called profiling and the resulting information is profiling data. The early code placement
techniques rely exclusively on summary profile data. This is organized as a call graph. The nodes are procedures, and the edge weights record the number of calls from a parent to a child procedure during a profiling run. See Chapter 3 for a more detailed discussion. The conflict metric derived from this data is limited in the amount of information it can convey and thus limits the effectiveness of code placement.

Gloy, Blackwell, Smith, and Calder [5] extend the concept of a profile to include a more precise measure of the amount of interleaving between procedures and use this extended information to formulate a new conflict metric described and improved upon in the next chapter.

2.2.2 Optimizing the Cost Metric

The second part of a code placement algorithm involves using the obtained metric to find a good ordering of the program code blocks. Most previous trace based techniques have been “greedy” in their approach [9, 8, 5]. The highest cost (according to the conflict metric) blocks are examined first. These high cost blocks are placed to avoid each other in cache, usually by mapping them to adjacent cache lines. All subsequent procedures are placed in the best spot relative to those already placed.

In Chapter 4, I present an alternative to the greedy approach using “local search” techniques. This results in a fast, robust algorithm which gives cache layouts as good as any other known code layout algorithm.

2.3 Benchmarks

The benchmarks used to test the algorithms in this thesis were obtained from Nick Gloy who used them to test his own code placement algorithm. A description of the seven benchmarks is shown in Table 2.1. A more detailed description of the training and test traces can be found in Nick Gloy’s thesis [6].

The focus of this thesis is on code layout and not on the accuracy of the training traces. So we assume that the training traces will give good predictions of the actual runs. Therefore we use the same traces for training and testing on all of our results.
Table 2.1: Benchmark Programs

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>File Size</th>
<th>Size of code in trace</th>
<th>Trace length (Train)</th>
<th>Trace length (Test)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>GNU C compiler</td>
<td>1432KB</td>
<td>556KB</td>
<td>241M</td>
<td>307M</td>
</tr>
<tr>
<td>go</td>
<td>'Go' strategy game</td>
<td>384KB</td>
<td>282KB</td>
<td>379M</td>
<td>609M</td>
</tr>
<tr>
<td>ghostscript</td>
<td>Postscript interpreter</td>
<td>528KB</td>
<td>161KB</td>
<td>93M</td>
<td>156M</td>
</tr>
<tr>
<td>latex</td>
<td>LaTeX typesetting application</td>
<td>312KB</td>
<td>134KB</td>
<td>153M</td>
<td>158M</td>
</tr>
<tr>
<td>perl</td>
<td>Perl language interpreter</td>
<td>352KB</td>
<td>73KB</td>
<td>398M</td>
<td>363M</td>
</tr>
<tr>
<td>porky</td>
<td>SUIF compiler code</td>
<td>1984KB</td>
<td>369KB</td>
<td>869M</td>
<td>965M</td>
</tr>
<tr>
<td>vortex</td>
<td>Object-oriented database</td>
<td>664KB</td>
<td>284KB</td>
<td>265M</td>
<td>516M</td>
</tr>
</tbody>
</table>

This allows to separate the effects of our placement algorithms from the potential inconsistencies of training and test data sets. However, comparable results are obtained in most cases when the testing traces described are used.

Each of the benchmark programs was compiled by Nick Gloy on a Digital workstation based on an Alpha 21164 processor running Unix 4.0. He used Digital’s Atom tool to generate the traces needed for profiling and memory simulation. A modified version of Nick’s profiling and memory simulation code was used to test our algorithms.
Chapter 3

Conflict Metrics and Heuristics

This chapter discusses the conflict metric used for trace based code layout. We briefly discuss the simple call graph based metric and then move on to the improved conflict metric proposed by Gloy, Blackwell, Smith, and Calder [5] and discuss its advantages as well as its limitations. We attempt to improve the model through a few heuristics and suggest other improvements and changes which might be the subject of future research. We finish with a presentation of methods to improve the accuracy of the metric via code transformations.

3.1 Conflict Metrics

The first part of a code placement algorithm is the extraction of information from the profiling traces. This information takes the form of a metric estimating the amount of cache conflict caused by any given layout. The problem of choosing a good metric is central to finding a good placement regardless of what algorithm is used for placement.

There are two requirements that the conflict metric must satisfy. The first, is that it should accurately reflect the number of cache misses for a give layout. The second is that it must be efficient to compute. The latter condition is due to the fact that an optimization algorithm will most likely have to compute the metric on many different layouts.

The ideal metric, of course, would be the exact number of cache misses resulting
for a layout on a particular profile set. In theory, this can be always be computed for any given layout once we have the instruction trace. Even ignoring the inevitable differences between the profiling data and the actual run time traces, this metric is undesirable since the time required to compute it seems to be proportional to the length of the trace. Therefore, we will consider conflict metric that merely approximate the number of conflict misses. If the approximation is good enough then the layout algorithms will perform well. In particular, the type of metrics we consider are pairwise metrics. For every pair of code blocks we assign a cost reflecting the penalty of these two blocks overlapping in the cache massing. The total cost of a layout is the sum of the pairwise costs for all the pairs of procedures that map to conflicting cache lines. More precisely, \[ \text{cost} = \sum_{ij} c_{ij} \cdot O_{ij}, \] where \( i \) and \( j \) range over the procedures, \( c_{ij} \) is the pairwise cost between \( i \) and \( j \), and \( O_{ij} \) is the number of overlapping cache lines for \( i \) and \( j \) in the given layout. Any pairwise metric can be represented as a weighted graph that we call the conflict graph, with code blocks as nodes and the pairwise cost as the edge between two nodes. Older cache placement algorithms [8, 9] used a metric based on a simple procedural call graph of the form output by standard profiling tools. Gloy, Blackwell, Smith, and Calder [5] developed a new metric which takes into account more data from the sample runs and is a more accurate predictor of cache misses.

### 3.1.1 Granularity

An important consideration in any code placement algorithm is the granularity of the objects being placed. This can be objects at the procedure level, the basic block level, or at some level in between. We will refer to the objects under consideration as code blocks. A courser granularity results in less space requirements for the conflict metric and a smaller search space for the code placement algorithm. A finer granularity results in a more accurate conflict metric and more freedom for placement.

**Procedures** In general the nodes of the conflict graph are code blocks at the granularity being considered. The larger the code blocks, the fewer nodes in the graph.
Most of the earlier code placement techniques used whole procedures as nodes. However, an implicit assumption in the cost formula above is that the amount of conflict between two code blocks is proportional only to the amount they overlap in cache. Thus it should be independent of what portions of the blocks are overlapping. This assumption is valid only if the blocks are uniformly executed. In practice this assumption rarely holds for whole procedures. Loops, jumps, and conditionals within the procedures result in some parts being executed far more often than others, and it is often the case that two overlapping procedures cause very few cache conflicts in one configuration but many conflicts in a slightly different configuration.

**Basic Blocks** Since cache conflicts occur between individual instructions, one way to maximize the accuracy of the conflict metric is to use individual instructions as nodes in the conflict graph. However, the space needed to store a pairwise conflict metric is $O(n^2)$ where $n$ is the number of nodes. So, the space requirements to store such a graph would be enormous, and the search space for finding a good placement would be far too large to apply any intelligent techniques. Fortunately, the finest granularity we need to consider is basic blocks. A basic block is defined to be a contiguously executed block of code delimited by branch instructions. Thus basic blocks always execute completely from start to finish and no information is lost by using basic blocks as nodes instead of individual instructions. Unfortunately the number of basic blocks is also often far too large to store a complete conflict graph. In the seven benchmark, medium sized, applications in this study, the number of basic blocks ranged from 10,000 to close to 80,000. This number can be reduced by considering only the most frequently executed basic blocks, but there are still several thousand and sometimes close to ten thousand relevant basic blocks. As the space to store the graph is quadratic, this results in a graph of size approaching several hundred megabytes. This number is large but still potentially feasible. However, another approach described below seems to give results nearly as good with a substantially smaller size.
**Chunk Level** An intermediate approach is to consider a granularity finer than whole procedures but larger than just basic blocks. A solution that works rather well is to consider approximately fixed size chunks consisting of one or more basic blocks. Gloy et al. came up with the idea of dividing procedures up into chunks of size averaging 256 bytes and built their conflict metric using these as nodes. 256 bytes is large enough so that almost no basic blocks are larger than a chunk but small enough to be mostly uniformly executed. Chunks are a much better approximation of the ideal, uniformly executed, blocks than procedures. Furthermore, unlike basic blocks, the number of chunks that contribute to the cache miss rate is only on the order of a few hundred to a few thousand resulting in a manageable size graph.

Note that Gloy et al. used chunks only as the granularity of the metric. Their placement algorithm operated on whole procedures, and chunks were used only as a method to better estimate the costs of partial overlaps of procedures. The local search algorithms we present later use chunks as the basic objects for both the metric and the placement. In the context of local search, placing chunks does somewhat better than placing whole procedures. In addition, we shall see that we obtain certain speed benefits from having all of the chunks close to the same size.

### 3.2 Call Graph Based Metric

A simple conflict metric can be obtained from the summary profile statistics provided by any standard profiler during a test run. Ordinarily the summary profile of a program execution contains just a weighted call graph or WCG that records the number of calls between pairs of procedures during that profiling run. Figure 3-1 contains an example call graph taken from GBSC. As described earlier, the conflict metric corresponding to a particular call graph assigns a penalty to any two procedures equal to the weight of the edge between them times the amount of overlap. The conflict cost for the graph is the sum of the penalties of all of the overlapping procedures.

The call graph provides a way of measuring the cost of putting two procedures on the same cache line. If procedure M makes many calls to procedure X then placing
Example of a program calling three leaf procedures. The call graph is obtained when the condition \( cond \) is true 50% of the time. Notice that both Trace \( #1 \) and Trace \( #2 \) produce the same call graph.

Figure 3-1: Call Graph Based Metric

M and X on the same cache line will certainly cause a lot of misses. However, the call graph based metric does not succeed in capturing all of the relevant information from the program traces. It only deals with first generation cache misses. Namely, it only looks at the conflict cost between a caller procedure and callee procedure. It does not provide any information about when these calls occurred or the extent of the dynamic interleaving with calls to other procedures. Consider the call graph in Figure 3-1. If the condition \( cond \) alternated between true and false we would obtain Trace \( #1 \), but if it were true 40 times and then false 40 times we would obtain Trace \( #2 \). Both traces lead to the same call graph. However, let us assume that we have only three lines in our cache, and all three procedures take one line each. Now, if one cache line is reserved for procedure M, in Trace \( #1 \) we are best off putting X and Y on separate cache lines and sharing Z with one of them. In Trace \( #2 \), on the other hand, we get fewer conflicts by putting X and Y on the same line and Z on the other line. So we see that the call graph does not fully capture the information needed to
best place the procedures.

3.3 GBSC metric

3.3.1 Basic GBSC Model

Notice that a cache line for a procedure \( p \) is reused exactly when no procedure using the same line is referenced between two calls to \( p \). On the other hand, if in between there is some procedure \( x \) using that line, we get a cache conflict when \( p \) is called the second time. This idea was exploited by Gloy, Blackwell, Smith, and Calder to create a new graph that better reflects the actual interleaving of procedures during a program trace. For each pair of procedures \( p \) and \( q \), the corresponding edge in a conflict graph is defined as the number of times we find \( q \) between two consecutive references to \( p \) or \( p \) between two consecutive references to \( q \). The resulting graph using Trace #2 from Figure 3-1 is shown in Figure 3-2. Gloy et al. refer to this graph as a *temporal relationship graph* or TRG. As the authors demonstrate, such a graph can be built fairly easily by processing the trace one code block at a time. Simply maintain an ordered list of all the referenced procedure identifiers. Notice that only the most recent reference to each procedure needs to be remembered since any future executions can only affect this most recent reference. Now, each time we encounter a procedure \( p \) which is not in the ordered list add it to the top of the list. Otherwise simply increment the edges \( e_{p,q} \) between \( p \) and all blocks \( q \) that have been referenced since the last reference to \( p \). Finally, update the ordered list so that this new reference to \( p \) is most recent.

Of course, the above idea is equally applicable using finer granularity than procedures. GBSC use chunks of size 256 bytes constructed by lumping together consecutive basic blocks in procedures. As an optimization they consider only the most frequently executed procedures for use in chunking and placement. The remaining procedures have very little effect on the miss rate and can be placed anywhere. Defining chunks as consecutive basic blocks exploits the spatial locality of programs to cre-
This is the graph resulting from the execution trace #2 given in 3-1. The graph shows that there is interleaving in this trace between Y and Z but not between X and Y

Figure 3-2: GBSC Metric

ate more uniform chunks than a pure random compilation might yield. A bit later we will discuss ways to further improve chunk uniformity through code transformations.

3.4 Refinements to the GBSC Model

This metric proves to be a dramatic improvement over the simple one derived from the weighted call graph. One drawback is that the precise interleaving information necessary is not provided by standard profiling tools. As mentioned above, it easy to extend a profiler to provide the extra information. Yet the cost of profiling the executable goes up, since for each executed block we must increment the edges of all blocks up until the last reference. Still, this cost is not prohibitive even for very large traces consisting of hundreds of millions of instructions or more. Note that like any pairwise metric, the GBSC metric is only a heuristic. It is certainly not a precise estimator of cache misses. Suppose two code blocks x and y occur in that order between successive references to p and all three share the same cache line. When p is referenced for the second time, it replaces y in the the cache but not x, since x has already been replaced by y. However, the above conflict metric counts an edge between both p and x and p and y. Therefore, the single cache miss for p is double counted.
3.4.1 Window Size

optimize Gloy et al. notice that if a sufficiently large amount of different code blocks have been executed between two references to a procedure $p$ then it is very likely that $p$ will have already been evicted from the cache by the time the second reference is reached. In theory, an optimized cache which minimizes the number of conflict misses should try to map the procedures between consecutive references of $p$ to different cache addresses. Therefore, when the distance (in unique code blocks) between the references is about the size of the cache, the first reference no longer contributes to the interleaving. This idea can be incorporated into the TRG construction algorithm by maintaining a bound on the size of the ordered list used to increment the edges.

Figure 3-3: Effects of Window Size on the L1 Cache Miss Rate

Figure 3-3 shows the effect of window size on the cache miss rate. The metric is recomputed for each value of the window size and the simulated annealing algorithm from Chapter 4 is used to this metric. Similar results can be obtained using other code layout algorithms. The window size of the metric, measured in multiples of the cache size, is plotted against a normalized average miss rate for an 8KB, 32 bytes per line, instruction cache. For each of the seven benchmarks, the miss rate is normalized so that the rate at a window size of 0 is equal to 1. It appears that a window size above a certain threshold value does not improve the cache miss rate. If the window size is too small the miss rate is high since not all the conflict is captured. Increasing the window size beyond this cutoff rate neither improves nor decreases miss rate.
From the graph we see that window size of twice the cache size is sufficient to capture all of the conflict. This suggests that changing the window size does not help much with the double counting problem. Although we certainly eliminate counting some irrelevant cache misses, the relative values of the edges do not change much. All of the edges in the graph are reduced in the same proportion. However, a smaller window size improves the time and space requirements of the trace processing algorithm. The run time of the trace processing algorithm is at most the window size times the trace length, since at worst we have to increment an edge for every reference in the window. So, reducing the window size improves performance by speeding up the trace processing.

3.4.2 Probabilistic Metric

The metric described above adds 1 to the edge weight for any interleaving of two blocks at a distance of up to the window size apart in the trace. A more accurate count might be obtained by somehow weighting the interleaving so that interleavings close together contribute more to the cost metric. If $p$ closely follows $x$ in the trace, then it is more likely to actually replace $x$ in cache then if there was a larger distance between $p$ and $x$ (measured in unique code bytes executed) in the trace. A first pass at implementing this idea is to assume that the chunks will be distributed uniformly at random in cache. Now, if $l$ cachelines of unique code occurs between chunks $p$ and $x$, then the probability a cache line containing $x$ has not been replaced by the time $p$ appears is \( \left( \frac{\text{cache_lines} - 1}{\text{cache_lines}} \right)^l \). Thus we can add this value to the edge instead of adding 1 for every interleaving. However making this change to the metric does not seem to significantly affect the code layouts. Part of the reason may be that the code blocks are not actually placed at random. If $p$ is a block that will be referenced again, then in a good code placement algorithm it is more likely that $p$ will still be in the cache after $l$ lines then random chance might dictate. In an attempt to incorporate this effect into the metric, we added a parameter $\alpha$ so that the value \( \left( \frac{\text{cache_lines} - 1}{\text{cache_lines}} \right)^\alpha \) is added to the metric at each stage. The implication is that the probability of a chunk occurring between references to $p$ mapping to the same line as $p$ is changed from
$\frac{1}{\text{cache lines}}$ to approximately $\frac{\alpha}{\text{cache lines}}$. Since a good layout algorithm should reduce the amount of interleaving that occurs, intuition suggests that an optimal value for $\alpha$ lies between 0 and 1.

Figure 3-4: Effects of Alpha on the L1 and L2 Miss Rates

Figure 3-4 shows how the parameter $\alpha$ affects the L1 and L2 miss rates for an 8KB L1 cache and 64KB L2 cache. Once again all the benchmarks are plotted on the same graph and the miss rate values are normalized so that the value at $\alpha = 0$ is 1. The effect of $\alpha$ on the L1 cache miss rate is not as significant as we might have hoped for. There is no more than a 5% difference for any of the benchmarks in the range of values considered. Surprisingly, increasing $\alpha$ above 1 seems to actually improve the L1 miss rate. However, this is far outweighed by the significant worsening of the L2 miss rates at higher values of $\alpha$. Although the data is a bit noisy, there appears to be a local minimum for both L1 and L2 miss rates at around $\alpha = 0.5$.

A possible explanation for the results may lie in the changing working set size during the trace. With a small working set, we would like a higher conflict cost to chunks very close together since they will cause the most conflict. This may be the cause of the slight improvement in the L1 miss rate. (It is only a slight improvement since chunks in the same working set will interleave a lot and thus not very likely to be mapped to the same line even with a smaller $\alpha$) On the other hand, if the working set is a bit larger than the cache size than any placement will result in misses in the
L1 cache. However with a high value of $\alpha$ the cost for putting these blocks on the same line is reduced, so a larger proportion of this conflict could take place in the same cache line or set of cache lines. This makes it harder to separate the chunks in these cache lines in the L2 cache resulting in the higher L2 miss rates.

### 3.4.3 Other Metric Refinements

The basic metric devised by Gloy et al. combined with the heuristic improvements described above seem to give a fairly good prediction of the number of cache misses, and work well with the local search based placement algorithm described below as well as the older greedy approaches. However, there are still additional modifications one could make that might give additional improvements. One idea is to adaptively change the edge weights of the TRG based on the actual layouts obtained. For example, compute a number of layouts for a given TRG and then compute the average number of actual cache conflicts for each pair of cache chunks. Then update the TRG by increasing the edges for the high conflict blocks, while decreasing the edges for those blocks who map to the same line but do not have as much conflict as the metric suggests. Continue in this manner updating the TRG and computing new layouts until some sort of equilibrium is reached. The resulting layouts might be better than at any of the intermediate stages.

On a completely different note, one could consider a metric incorporating triplets of chunks or larger sets of chunks. Unfortunately the space requirements for maintaining this metric grows exponentially with the size of the sets. Perhaps some means of using mostly pairwise information but occasional higher set information could be devised.
3.5 Improving the Metric through Code Transformation

Another potential source of improvement lies in the formation of the chunks themselves. Creating more uniformly executed chunks results in a more accurate cost metric as discussed above. Below we discuss some code transformation techniques that help improve chunk uniformity: fluff separation and code linearization.

3.5.1 Separating Fluff

The old rule of thumb that 10% of the code uses 90% of the run time applies here in force. In fact, for the example traces used here, two thirds or more of the basic blocks can be disregarded and the remaining still makes up over 99% percent of the executed instructions and 99% of the total cache misses. The rest of the basic blocks are essentially “fluff” and can be placed arbitrarily without having too much effect on the cache miss rate. Often times fluff basic blocks are interspersed with frequently executed code in the form of a rarely executed conditional. Separating out the fluff basic blocks before dividing procedures into chunks results in more uniform chunks. Once the rare conditional is removed, the rest of the code is usually executed more or less sequentially.

A simple algorithm for fluff separation is as follows. Sort all basic blocks in descending order of frequency of execution. Now find all basic blocks which together make up most (99.9% in these examples) of the execution time. Mark all other basic blocks as fluff. Now when constructing the chunks, do it as before by iterating through the procedures and adding consecutive basic blocks until the target chunk size is reached, except this time skip over any fluff basic blocks.

After marking the fluff code, many rarely executed procedures will be entirely fluff. There is no point in dividing these procedures into chunks. Instead, only place the procedure chunks containing non-fluff code (these are the popular procedures) and use the rest of the unpopular procedures to fill in gaps in the final layout between
the popular code chunks [5]. The fluff basic blocks that were part of the popular procedures can either also be used as fillers or simply all placed at the end of the code.

An alternate way to perform fluff separation is to select the most popularly executed procedures (instead of basic blocks), and then mark as fluff those basic blocks within the popular procedures that are rarely executed in the sample trace. The infrequency at which a basic block is executed before it is called fluff could depend on the frequency of its parent procedure, however simply marking all blocks which are never executed in the sample trace works just as well. This is the method implemented by Nick Gloy in the GBSC algorithm and described in his Ph.D. thesis [6]. Since the popularity of basic blocks drive the conflict misses and not so much the popularity of their corresponding procedures, this method should be slightly less accurate.

Table 3.1 below compares the primary (L1) cache miss rates of fluff separation techniques. In all cases the chunk based simulated annealing algorithm from Chapter 4 is used to find the code layouts. The first column labeled 'No Fluff' uses only popular procedure selection and does not separate out fluff code from within popular procedures. The second column labeled 'Gloy Fluff' uses the fluff separation algorithm from Nick Gloy’s Ph.D. thesis. Only popular procedures are selected and basic blocks that are never executed in the trace are placed at the end. The third column 'New Fluff' uses the popular basic block driven fluff separation described above.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>L1 Cache Miss Rate</th>
<th>% Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NoFluff</td>
<td>Gloy Fluff</td>
</tr>
<tr>
<td>gc</td>
<td>5.005</td>
<td>3.991</td>
</tr>
<tr>
<td>go</td>
<td>1.528</td>
<td>1.470</td>
</tr>
<tr>
<td>gs</td>
<td>1.474</td>
<td>1.160</td>
</tr>
<tr>
<td>la</td>
<td>1.627</td>
<td>1.303</td>
</tr>
<tr>
<td>pl</td>
<td>1.950</td>
<td>0.439</td>
</tr>
<tr>
<td>po</td>
<td>2.587</td>
<td>2.291</td>
</tr>
<tr>
<td>vo</td>
<td>4.460</td>
<td>2.872</td>
</tr>
</tbody>
</table>
The results demonstrate the dramatic improvements as a result of fluff separation. On average Gloy’s fluff separation is 27.1% better than just popular procedure placement, while the new popular basic block selection is 30.2% better.

### 3.5.2 Code Linearization

Fluff separation is a special case of the more general technique of code linearization. This means reordering the basic blocks so as to maximize the number of times consecutive blocks in the code are executed consecutively in the program trace. In addition to increasing the uniformity of the chunks, code linearization improves the cache miss rate when two basic blocks share a cache line. A miss on one block will also load the other. If they are usually executed consecutively this can reduce the number of misses due to the prefetching. Code linearization has been investigated in the context of branch alignment by [10, 8]. While some useful techniques have been developed, they are time consuming and just as complicated as the code layout problem itself. Therefore, a full code linearization is most likely overkill when just optimizing for cache misses. The fluff separation heuristic described above is probably sufficient in practice.
Chapter 4

Local Search Algorithms

In this chapter we apply the paradigm of local search to code placement algorithms. The basic idea behind local search algorithms is to start with an initial layout and make local changes until an optimum is reached. This results in robust algorithms giving results comparable to any in the literature with an asymptotically faster run time.

4.1 Granularity of Placement

Most work on code placement has involved the placement of whole procedures. Gloy, Blackwell, Smith, and Calder came up with the idea of using a fixed size chunk. A chunk is composed of consecutive basic blocks from within a procedure up to a target size of 256 bytes. This is small enough so that the metric is accurate but large enough so that it is larger than almost any individual basic block. However, even GBSC use the chunks only to construct a finer granularity metric to determine the best offsets of procedures. They still place whole procedures in a greedy manner. The local search algorithms we present below are not constrained to keep the procedure chunks together. Instead all of the chunks are placed independently. These chunks are all approximately the same size (they are not exactly the same size since they contain an integral number of basic blocks). This allows to use the following result:

**Lemma 1** Consider a circular cache with $L$ lines $0, 1, \ldots, L - 1$. Suppose there are
n code blocks each of size exactly \( l \) lines with \( l \) dividing \( L \). Let \( o_1, o_2, \ldots, o_n \) be the offsets of each of the blocks in the cache. Let \( O_{ij} \) be the overlaps of blocks \( i \) and \( j \) in cache. Now there is a minimum for the pairwise metric \( \sum_{i,j} c_{ij} O_{ij} \) such that each \( o_i \equiv 0 \mod l \).

Proof: We define a frame \( F_k \) of blocks to be the set of all blocks whose offset is congruent to \( k \mod l \). We show that we can find an optimum in which all of the frames are aligned. Consider a particular frame \( F_k \). Choose a chunk \( i \in F_k \). For every chunk \( j \notin F_k \) overlapping \( i \), moving \( i \) to the left one cache line will cause either a decrease in the overlap with \( j \) by one line or an increase in the overlap by one. Moving \( i \) to the right one line will cause the opposite. Hence, the net change in the cost function for these two chunks due to a movement in one direction is the negative of the net cost due to a movement in the other direction. Summing over all procedures \( i \in F_k \) and all procedures \( j \notin F_k \), we see that the net change in the cost function due to the movement of the entire frame \( F_k \) to the right one line is the negative of the cost of moving it to the left one line. Thus at least one of these movements is non-negative. We can move \( F_k \) in that direction until it is aligned with some other frame \( F_l \) and not increase the cost function. Therefore, we have reduced the number of non-aligned frames by one. Continuing in this manner, we can align all of the frames. Without loss of generality, assume the frames are aligned at the \( 0 \) offset modulo \( l \).

As a consequence of the above lemma see that if the chunks are all the same size then when searching for a minimum placement we need only consider positions for the chunks that are a multiple of the chunk size. This drastically reduces the size of the search space since there are only \( \frac{L}{l} \) positions for each chunk whereas originally there were \( L \) positions.
4.2 Local Optimization

4.2.1 Layout and Move Representations

Every local search starts with an initial placement of the code blocks, which a priori can be at any level of granularity, and tries to improve this placement via local moves to get new candidate layouts until either no more improvements are possible, or the allocated search time is exhausted. For the code layout problem, a candidate layout is an assignment of code blocks to cache offsets. It can represented by a single array offset where offset[b] ∈ {0, 1, ..., cachelines − 1} is the starting offset (in cache lines) for the code block b. Later we describe how to convert an offset description such as this into a linear ordering of the code blocks that can be output by the compiler. Note that if the size of b is more than one cache line long then the cache lines that the block b occupies are offset[b], offset[b] + 1, ... . These values are taken modulo the total number of cache lines in the cache, as a direct mapped cache maps addresses modulo the cache size. A local move corresponds to changing the offset value for a single block b.

4.2.2 Computing the Cost Function

Recall that the cost metric is defined as \(\sum_{ij} c_{ij} O_{ij}\) where i and j range over the code blocks, \(c_{ij}\) is the edge between i and j in the TRG, and \(O_{ij}\) is the amount of overlap between blocks i and j. In order to implement a local search algorithms we need to be able to efficiently compute the cost metric for a given layout. More precisely, we need to be able to efficiently compute the change in the cost function with respect to a local move (i.e. a movement of a single block b from one offset to another). To do this we associate to a layout another data structure cachelines, so that cachelines[i] is a list of the block references that occupy cache line i. Thus the overlap is \(O_{ij} = |\{l : i, j \in \text{cachelines}[l]\}|.\) Now, the main subroutine necessary for cost computation is \texttt{Cost.Block.Offset} which computes the cost for a single block at any offset given a placement of all the other blocks. The pseudocode for this
Figure 4-1: Pseudocode for Cost Computation

// Compute cost of putting block b in location loc
Cost_Block_Offset(int b, int loc)
{
    cost = 0;
    for (l = loc; l < loc + size(b); l++)
    {
        foreach b2 in cachelines[l]
            if (b2 != b)
            {
                cost += weight_on_edge(b, b2);
            }
    }
    return cost;
}

// Compute total cost with block locations stored in offset
Compute_Total_Cost(offset)
{
    cost = 0;
    foreach b in blocks
    {
        cost += Cost_Block_Offset(b, offset[b])/2;
    }
    return cost;
}

subroutine and the total cost computation routine is shown in Figure 4-1.

Note that when computing the total cost we sum the cost for each block, and then divide by 2. This is since the overlap for each pair of procedures is counted twice, once for each procedure. Now, the incremental cost of moving a block b from location l1 to location l2 is just Cost_Block_Offset(b, l2)−Cost_Block_Offset(b, l1). Note that the code above works for arbitrary size blocks.\(^1\)

Furthermore, in the special case that all the blocks are chunks, which are the same size and divide the size of the cache, we can make some simplifications. The alignment result from above allows us to assume that all chunks overlap completely or not at all. Thus, the only possible offsets for a given block are multiples of the

\(^1\)A few minor technical modifications have to be made when dealing with blocks that are larger than the cache size, this is not difficult and is not shown here

34
chunk size. Furthermore, the cost due to a chunk at a given offset depends only on 
the blocks at that offset and no others. This allows us to simplify the code as shown 
in Figure 4-2. Figure 4-2: Simplified Cost Computation for Chunks

```c
// Compute cost of putting chunk c in location loc
Cost_Block_Offset(int c1, int loc)
{
    cost = 0;
    foreach c2 in cachelines[loc] //all chunks in loc
    {
        if (c2 != c)
        {
            cost += chunksize * weight_on_edge(c, c2);
        }
    }
    return cost;
}
```

Notice that we no longer need the outer loop in Cost_Block_Offset.

4.2.3 Local Optimization Algorithm

A very simple local search algorithm is to start with a random placement of chunks 
and iterate through the chunks moving each to the best possible offset. At each stage 
the total cost is non-increasing. To avoid an infinite loop we arbitrarily choose the 
smallest offset when faced with a choice of equally good best offsets. At some point 
we must reach a layout that cannot be improved anymore through local moves, in 
other words each chunk is in the best possible spot when holding all other chunks 
fixed. Such a position is called a local optimum. The algorithm described above is a 
local optimization algorithm. Pseudocode for local optimization is shown below.

Note that we chose an initial placement where each chunk is mapped to a random 
offset. This is better than using a fixed initial placement since it makes the layout 
independent of the default layout. So, small changes to the initial layout, such as 
adding one line of code to one of the procedures, should not have a large effect on 
the corresponding local optimized cost function and thus on the cache miss rate.

35
// initial random placement
for (i = 0; i < num_chunks; i++)
{
    offset[i] = random 1 in {0, CL, 2*CL, ... (k-1)*CL}
    // where CL = # of lines in a chunk, and k = #_cache_lines/CL
}

flag = TRUE;
cost = 0;
cost = Compute_Total_Cost(offset);

// Repeat until no more improving moves

while (flag)
{
    flag = FALSE;
    for (c = 0; c < num_chunks; c++)
    {
        new_cost = INFINITY;
        foreach cl in (0, CL, 2*CL, ... , (k-1)*CL)
        {
            temp_cost = Cost_Block_Offset(c, cl);
            if (temp_cost < new_cost)
            {
                new_offset = cl;
                new_cost = tcost;
            }
            if (cl == offset[c]) //store the old_cost of c
                old_cost = tcost;
        }
        if (new_offset != offset[c])
            FLAG = TRUE;
        delete c from cache_lines[offset[c]];
        offset[c] = new_offset;
        add c to cache_lines[new_offset];
        cost = cost + new_cost - old_cost;
    }
}
4.2.4 Drawbacks

Local optimization turns out to be a good, fast algorithm for minimizing the objective function. The highest cost conflicts are very quickly separated and in even the largest cases convergence to the local minimum occurs after a small number of iterations. However there are some drawbacks to simple local optimization. In particular, the algorithm will always stop as soon as it reaches a local minimum. But there is no guarantee that this local minimum will be near the true minimum. In theory, it is possible to get trapped in a very poor local minimum. One possible solution is to run local optimization multiple times and take the best solution obtained. This is a waste of effort since we would be repeating the work taken to get down close to the minimum multiple times. A more robust algorithm would provide some way to escape from poor local minimums and thus never be stuck with a poor solution and always be closer to the real minimum.

4.3 Simulated Annealing

Simulated annealing is a heuristic for running local searches while avoiding poor quality local optima. Just as in local optimization, we start with some initial solution and try to improve upon it using local moves. The difference is that under simulated annealing we not only make moves that decrease the objective function, but with some probability accept moves that actually increase the objective function. The motivation for simulated annealing comes from cooling systems in physics. Metropolis et al. [11] developed a method of simulating energy levels in a cooling solid. Any perturbation which results in a decrease in energy level is always accepted. However, a perturbation which increases the energy level by an amount $\delta$ at a given temperature $T$ is accepted with probability $e^{-\frac{\delta}{kT}}$ where $k$ is Boltzmann’s constant. This formula is known as Boltzmann’s law in physics. The temperature starts at some large value and gradually decreases in a fixed way until the system no longer changes much (i.e. is “frozen”). Notice that as $T$ decreases the probability of accepting an uphill perturbation decreases.
In 1982, Kirkpatrick et al. [12] adapted the simulated annealing model for combinatorial optimization problems. The energy level corresponds to the objective function that is desired to be minimized. The perturbations are local moves. Thus, any problem with a "neighborhood" structure on the solution space (i.e. a set of transitions from any solution to a set of adjacent solutions), and an efficient method of computing the change in the objective function with respect to a transition, can be by simulated annealing. Since then a number of authors have attempted to adapt simulated annealing to various kinds of combinatorial optimization problems. While the method itself is very general a number of implementation details and parameters must be set for the problem at hand. Below we discuss how this was done for code placement.

4.3.1 Scheme Outline and Notation

The basic annealing scheme is as follows. The temperature starts at some sufficiently high value to allow the higher conflict blocks to be able to settle in to a good position. At each temperature the simulation is run for a certain amount of time accepting moves according to Boltzmann's law as described above. The temperature itself is slowly decreased. As the temperature decreases, the high conflict blocks stabilize and so only the lower conflict blocks move. At a sufficiently low temperature the system is considered frozen and the simulation stops. At the end of the simulation the best solution is output.

Table 4.1 describes the parameters and quantities used in the description of the various components of the annealing scheme described here.

4.3.2 Single Temperature Run

The most important subroutine for simulated annealing is the routine which performs the search at a single fixed temperature. The number of iterations this subroutine runs is Epoch-Length. If \( n \) is the number of blocks and \( k \) the number of possible offset positions for \( n \), then the number of neighbors \( N \) for a layout is \( nk \). The Epoch-Length
Table 4.1: Simulated Annealing Quantities and Parameters

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Phi$</td>
<td>The cost function</td>
</tr>
<tr>
<td>$\delta$</td>
<td>Incremental change to cost function after a move</td>
</tr>
<tr>
<td>$T$</td>
<td>Temperature</td>
</tr>
<tr>
<td>$e^{\frac{\delta}{T}}$</td>
<td>Probability of making an uphill move</td>
</tr>
<tr>
<td>$Start_Temp$</td>
<td>Starting Temperature for annealing</td>
</tr>
<tr>
<td>$End_Temp$</td>
<td>Ending Temperature for annealing</td>
</tr>
<tr>
<td>$Temp_Ratio$</td>
<td>Ratio of $Start_Temp$ to $End_Temp$</td>
</tr>
<tr>
<td>$Temp_Factor$</td>
<td>Rate of cooling</td>
</tr>
<tr>
<td>$n$</td>
<td>Number of chunks to be placed</td>
</tr>
<tr>
<td>$N$</td>
<td>Number of neighbors of a candidate solution</td>
</tr>
<tr>
<td>$Epoch_Length$</td>
<td>Basic unit of simulation length</td>
</tr>
<tr>
<td>$L$</td>
<td>Total simulation length</td>
</tr>
</tbody>
</table>

is some multiple of $N$. Here we use $Epoch\_Length = 2N$.

The subroutine attempts to make each of $Epoch\_Length$ moves at the given temperature $T$. The move are chosen by cycling through all of the blocks according to a permutation of the $n$ blocks. Each block is attempted to be moved to a random offset location. After the cycle of $n$ blocks is complete, the permutation is shuffled and the algorithms cycles through this new permutation. (See pseudocode below) Another strategy is to choose both the block and the offset at random for each move. However, Connoly [27] showed that sequential search tends to do somewhat better than pure random search since we are more likely to cover all of the branches in the search tree. Taking this idea even further one could sequentially try all of the positions for one block before moving on to the next block. This method however is far too close to the ordinary “move to best spot” heuristic of ordinary local optimization and tends to fall into local optima more often. The cycle of blocks is shuffled after each pass to avoid falling into patterns.

4.3.3 Temperature Control

We implemented a very simple exponential cooling schedule for the simulated annealing. The temperature is multiplied by a constant $Temp\_Factor$ at each cooling stage.
Figure 4-4: Simulated Annealing at One Temperature

One_Temp(temp)
{
    for (i = 0; i < Epoch_Length; i++)
    {
        index = i mod num_chunks;
        if (index == 0)
        {
            shuffle_array(chunks);
        }
        place = random in {0, CL, 2*CL, ..., (k-1)*CL}
        c = vertices[index];
        old_cost = Cost_Block_Offset(c, offset[c]);
        new_cost = Cost_Block_Offset(c, place);
        delta = (new_cost - old_cost);
        exp_val = exp(-1*delta/temp);
        // Accept 0 cost moves with 50 percent probability
        if (delta == 0.0)
            exp_val = 0.5;
        r = frandom();
        if ((move = (r < exp_val)) & (delta != 0.0))
            accepts++;
        total++;
        if ((delta < 0.0) || move)
        {
            delete c from cache_lines[offset[c]];
            offset[c] = place;
            add c to cache_lines[place];
            current_cost = current_cost + new_cost - old_cost;
        }
    }
    if (current_cost < best_cost)
    {
        best_cost = current_cost;
        best_temp = temp;
        for (i = 0 ; i < num; i++)
        {
            best_offset[i] = offset[i];
        }
    }
}
The temperature starts at some value \( \text{Start\_Temp} \) and is reduced to \( \text{End\_Temp} \). Suppose the total number of iterations is \( L \). Then the relationship between these quantities is \( \text{End\_Temp/Start\_Temp} = (\text{Temp\_Factor})^L \).

**Choosing End\_Temp**  Clearly \( \text{End\_Temp} \) should be the value at which the system is frozen. If this is made too large, then the system does not fully settle. If it is too small, then search time is being wasted at temperatures where very few moves are taking place. The best value of \( \text{End\_Temp} \) depends on how large the objective function is. Our method for choosing \( \text{End\_Temp} \) is to run the search a small number of iterations at a very low temperature to find the typical value \( \Phi \) of the objective function. Then set \( \text{End\_Temp} = 2 \times 10^{-6} \cdot \Phi \). The value \( 2 \times 10^{-6} \) was determined empirically and corresponds to accepting a move which increases the objective function by 0.0002% with a probability of 0.38.

**Choosing Start\_Temp**  The starting temperature is chosen to be some multiple \( \text{Temp\_Ratio} \) of the \( \text{End\_Temp} \). For a simulation of fixed length \( L \) (measured in number of calls to the OneTemp routine) these quantities determine the cooling ratio by the formula \( \text{Temp\_Factor} = \text{Temp\_Ratio}^{\frac{1}{L}} \). The following graphs in Figure 4-5 show the results of varying \( \text{Temp\_Ratio} \) at fixed simulation lengths of 5, 10, 25, 50, 100, and 200. The average final value for the conflict metric is plotted against the temperature ratio. Each graph represents one benchmark and there are separate curves for each of the simulation lengths. For every simulation length and for all five benchmarks, the average final value of the metric drops sharply with increasing \( \text{Temp\_Ratio} \) until a value of 512, at which it stabilizes. The optimal value for \( \text{Temp\_Ratio} \) does not seem to depend on the simulation length. At the shorter simulation lengths, the search results start to worsen for values of \( \text{Temp\_Ratio} \) larger than 512.

The rationale is that the starting temperature needs to be high enough to allow the high conflict blocks to move around and settle into a good spot. Increasing it above this amount does not help since the moves start looking more and more random at higher temperatures. For the smaller simulations, too high temperatures result in
very fast cooling, which does not let the high conflict blocks settle into a good spot. Longer simulations do not change the optimal temperature but can tolerate higher than optimal starting temperatures without affecting the search results too much.

4.3.4 Simulation Length vs. Quality Tradeoff

The graphs also demonstrate increasing solution quality with increased simulation length. The next set of graphs in Figure 4-6 isolate this relationship at the optimal temperature ratio of 512. The simulation length is plotted against the average final conflict metric value. We see that the gain due to increased search time drops off as the search time increases. Changing the search length from 5 to 10 or 20 has a much larger effect then changing it from 100 to 200. From the graphs we see that a search length of 100 is sufficient to capture almost all of the benefit that can be gotten from simulated annealing. Even a search length of 50 might be preferred for efficiency reasons.

Table 4.2 compares simulated annealing at a search length of 200 with ordinary local optimization. Simulated clearly finds better quality solutions, however even for a search length of 200 the gain is only a $1 - 3\%$ improvement in the metric. The problem is that the solution space contains many, many local optima all with about the same value. Furthermore, these local optima are all qualitatively very different. The set of chunks which cause the most conflict varies drastically from one solution to the next, yet the total cost comes out about the same. Simulated annealing succeeds in penetrating the very top layer of local optima in order to find solutions that are a few percent better, however in the end there are too many local optima for it to escape completely and converge on a significantly improved solution if it exists. It is also possible that no significantly improved solution exists.

4.3.5 Complexity Analysis

Both the temperature selection and simulation portions of the annealing require at most a constant number of calls to \texttt{One.Temp}. Thus the asymptotic time bound is the
Table 4.2: Comparison of Local Optimization and Simulated Annealing

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Local Opt.</th>
<th>Sim. Anneal.</th>
<th>% Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td>$7.292 \times 10^5$</td>
<td>$7.207 \times 10^5$</td>
<td>1.2</td>
</tr>
<tr>
<td>gs</td>
<td>$4.589 \times 10^6$</td>
<td>$4.511 \times 10^6$</td>
<td>1.7</td>
</tr>
<tr>
<td>la</td>
<td>$6.844 \times 10^6$</td>
<td>$6.613 \times 10^6$</td>
<td>3.1</td>
</tr>
<tr>
<td>pl</td>
<td>$3.686 \times 10^6$</td>
<td>$3.632 \times 10^6$</td>
<td>1.5</td>
</tr>
<tr>
<td>vo</td>
<td>$3.221 \times 10^6$</td>
<td>$3.112 \times 10^6$</td>
<td>3.4</td>
</tr>
</tbody>
</table>

same as the time spend in this function.

For the analysis we assume $Epoch\_Length$ is twice the total number of adjacent moves $N$. Now $N$ is just the number of blocks $n$ times the number of offset positions $k$. During each of the $Epoch\_Length$ iterations we do a constant amount of work to pick a random move, decide whether to make the move after computing the probability, and if necessary make the move. The rest of time is taken up by two calls to $Cost\_Block\_Offset$. One of the two calls computes the new cost of the candidate move. Since the offset of the move is chosen at random, there are an expected $n/k$ blocks at the new offset. Thus it takes $O(n/k)$ time to compute the new cost. The other call to $Cost\_Block\_Offset$ is to compute the old cost and takes time proportional to the number of blocks in the current position. In the worst case this is $O(n)$, however if, as expected, the blocks are distributed evenly this time is also $O(n/k)$. Hence, the total time is $2nk \cdot n/k = O(n^2)$. On the other hand, the greedy algorithms $PH$ and $GBSC$ have been shown to require $O(P^3)$ time where $P$ is the number of procedures. Unless there is a small number of very large procedures, our method will be asymptotically much faster than either of the greedy algorithms. Furthermore, the size of the metric itself is $O(n^2)$, thus we require at least $O(n^2)$ time to even read in the data. Therefore, our algorithm has an optimal asymptotic runtime for any placement algorithm based on a pairwise chunk metric.

Local optimization is a bit trickier to analyze since we cannot predict how many iterations are necessary until a local minimum is reached. In practice this is rarely more than 10 or 20, although it seems to grow very slowly with the size of the problem.
An enforced upper bound of 50 will guarantee a $O(n^2)$ running time. In practice local optimization is the fastest of all the algorithms.

Table 4.3 shows the average run time in seconds of local minimization (LM), simulated annealing with a search length of 50 (SA), GBSC, and the Pettis and Hansen algorithm (PH) on seven benchmark applications. The data was obtained using the unix time command on an Pentium Pro 200 with 256MB of RAM running the LINUX operating system.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Procs</th>
<th># Chunks</th>
<th>Average Running Time in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>gc</td>
<td>743</td>
<td>1867</td>
<td>11.06, 227.94, 1302.37, 55.05</td>
</tr>
<tr>
<td>go</td>
<td>254</td>
<td>781</td>
<td>3.66, 37.72, 122.19, 4.24</td>
</tr>
<tr>
<td>gs</td>
<td>412</td>
<td>584</td>
<td>2.52, 22.41, 129.51, 19.88</td>
</tr>
<tr>
<td>la</td>
<td>183</td>
<td>450</td>
<td>1.56, 12.77, 6.95, 1.45</td>
</tr>
<tr>
<td>pl</td>
<td>59</td>
<td>127</td>
<td>0.69, 2.90, 6.95, 1.45</td>
</tr>
<tr>
<td>po</td>
<td>900</td>
<td>1406</td>
<td>7.22, 136.58, 809.93, 157.52</td>
</tr>
<tr>
<td>vo</td>
<td>265</td>
<td>647</td>
<td>2.69, 27.99, 94.31, 10.78</td>
</tr>
</tbody>
</table>

The table shows that local optimization is clearly the fastest if all the algorithms. PH is remarkably fast for an $O(P^3)$ algorithm due to its extremely low overhead. SA with 50 iterations takes on average about 10 times as long as LM. Increasing or decreasing the search length will proportionally change the running time.

4.3.6 Intra-Block Optimization

Up until now we have been assuming all the chunks were exactly the same size. However, since chunks consist of an integral number of basic blocks, they may be slightly smaller than the actual chunk size. Also, the way the chunks were defined, they do not cross procedural boundaries. So at the end of each procedure there is a small leftover piece. Hence, if a set of chunks all map to the same set of lines in cache, some improvements can be obtained by moving around the chunks within that set of lines. One way to do this is through an additional intra-block simulated annealing,
but with such a restricted search space, local optimization does well enough for our purposes.

4.4 Extension to L2 Cache

We have so far developed a metric and algorithm for minimizing the conflict misses in the primary (L1) cache. Here we show how these methods can be extended to also optimize for the secondary (L2) cache.

4.4.1 Conflict Metric

The conflict metric for L2 cache misses is identical to the metric for L1 except the window size is extended to be proportional to the size of the L2 cache. As discussed with the L1 metric a window size of 2 and an alpha value of 0.5 works well to compute the metric.

4.4.2 Algorithm

Thus far in the L1 placement we have only specified the addresses of code blocks modulo the size of L1 cache. We have yet to specify any other information about their program addresses. Let $S_1$ and $S_2$ be the sizes of the L1 and L2 cache respectively. If $K = S_2/S_1$ and a block $b$ is at offset $o_b$ in L1 cache, then we can place it at any of the offsets $o_b, o_b + S_1, o_b + 2S_1, \ldots o_b + (K - 1)S_1$ in L2 cache without affecting the L1 miss rate. Hence we can use local optimization (or simulated annealing) to optimize for L2 miss rate on a given L1 placement. Now $\text{offset}[b]$ is extended to have two components $o_1$ and $o_2$. $o_1$ is the L1 offset position just as before, and $o_2$ is a number between 0 and $K - 1$ representing the L2 offset. The local optimization works separately on each chunk sized region of L1 cache (chunks overlap only with other chunks in their own region). A local move corresponds to a change in the value of the $o_2$ component of $\text{offset}[b]$ for a single chunk $b$. Computing the total L2 metric and incremental change to the L2 metric can be easily done similar to before. This
algorithm optimizes the L1 cache before optimizing the L2 cache. There are several reasons for doing this. First of all it is easier to find good placements in L2 cache due to its larger size and thus we are better off optimizing the hard to place L1 cache first. Second, L2 cache is often a unified instruction and data cache which means that instructions might be replaced by data and code placement has less of an effect.

4.5 Other Optimizations

Even after optimizing for L2 cache there is still a fair amount of freedom in choosing the addresses of chunks. Recall that the L1 cache offset space has been divided up into regions the size of a chunk. Overlaps between chunks happen only among chunks mapped to the same region and not among chunks in different regions. This provides us the freedom to reorder the regions in any way we please without affecting the miss rate. Furthermore, we can reorder the L2 blocks within each L1 region as well the chunks within each L2 block. One reason we might do this reordering is to improve the spatial locality of the code layout. Spatial locality refers to amount that chunks that are temporally related (i.e frequently executed close together in the trace) are also spatially correlated (physically close together in the memory mapping). Gloy points out in his Ph.D. thesis [6] that the spatial locality affects the virtual memory paging system. A spatial locality metric such as the one defined by Gloy can be used to do another simulated annealing or local optimization to reorder the blocks.

4.6 Creating the Linear Order

Once we have assigned offset positions to all of the chunks in the code, we need to construct a linear order of the chunks maintaining these offsets. The method we use is from GBSC [5]. Pick a chunk \( p \) whose offset is 0. Next add the chunk \( q \) whose cache-relative offset results in the smallest positive gap between the end of \( p \) and the start of \( q \). Continue adding chunks in this fashion until all chunks are exhausted. Whenever there is a gap between two chunks use the unpopular (and thus unplaced)
code to fill the gap. If there is any unpopular code remaining at the end, simply append the list of the unpopular procedures to the linear order.

4.7 Results

Table 4.4 summarizes the average cache miss rates for four code layouts, Default, P&H, GBSC, and Simulated Annealing (SA). All layouts have implemented fluff separation with all fluff basic blocks moved to the end of the layout. The values are computed for an 8KB L1 cache with 32 bytes cache lines and a 4 byte instruction length. The L2 cache is 64KB also with 32 byte cache lines.

For L1 cache, SA beats GBSC for five of the seven benchmarks. For L2 cache, the results are even more dramatic. SA algorithm is better than GBSC for six out of seven benchmarks, considerable better for four of them. Only on the pl benchmark does it do worse which might be a tradeoff due to the substantially better L1 miss rate on this benchmark.

More detailed results are available in Appendix A, where we plot each of the algorithms for 10 different runs. Since SA is a randomized algorithm we are interested in the variance of its outputs in addition to the average output. The graphs in the appendix plot 10 runs of each algorithm in sorted order. One algorithm outperforms another when all of its points are below in the graph. We see that SA has good performance in most cases in addition to its good average case behavior.

4.8 Comparisons to Associative Caches

One of the advantages of code layout algorithms such as SA and GBSC is that they achieve good performance using only direct mapped caches. In this section we compare the performance of the SA algorithm to fully associative caches. In particular we compare it associative caches using LRU and Belady's optimal caching algorithm [13]. Code mapping algorithms work by using knowledge of the future to find a good mapping. Belady's algorithm has perfect knowledge of the future to find the optimal
replacement. We find that associativity and knowledge of the future about equally important in getting good cache miss rates. In particular SA code placement does almost as well as a fully associative LRU cache. Both algorithms do substantially worse than Belady’s which is able to exploit both properties. An additional factor, code linearity, is also very important for caches with more than one instruction per line. We also show some hardness results which show that it is impossible to find an optimal cache layout algorithm for direct mapped caches efficiently unless $P = NP$.

4.8.1 LRU cache

The most common replacement strategy for associative caches is Least Recently Used (LRU). This algorithm replaces the oldest cache line every time a new line is loaded into the cache. LRU is the de facto standard for cache replacement strategies and has
often been used as a benchmark measure of optimal caching. (For example, conflict misses were originally defined as the number of misses more than a fully associative cache of the same size using LRU replacement). Table 4.5 compares the average L1 cache miss rates of SA and fully associative LRU. The LRU miss rate is computed for both the standard layout and the standard layout with fluff separation (the effect of this on LRU caching is described below).

Using only a direct mapped cache and the SA placement algorithm we outperform the default fully associative LRU on five out of seven benchmarks, and by an average of 19.6%. However we find that this improvement is due mainly to the fluff separation portion of the algorithm. As discussed before, fluff separation increases the linearity of the code and improves the cache miss rate by increasing the average of number of instructions per cache line executed. Nevertheless, even comparing both fluff separated rates, the direct mapped SA cache is only 9.5% percent above the fully associative rate. This is a very small price to pay for the other benefits of direct mapped caching.

### 4.8.2 Belady’s algorithm

If a cache is fully associative and the trace is known than there is an algorithm to implement the *optimal* cache replacement strategy. This is known as Belady’s furthest in future algorithm [13]. The algorithm is to simply always evict the cache object
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>OPT Miss Rate</th>
<th>SA Miss Rate</th>
<th>% Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>gc</td>
<td>11.980</td>
<td>22.120</td>
<td>+84.6</td>
</tr>
<tr>
<td>go</td>
<td>7.051</td>
<td>9.490</td>
<td>33.8</td>
</tr>
<tr>
<td>gs</td>
<td>2.530</td>
<td>6.094</td>
<td>140.9</td>
</tr>
<tr>
<td>la</td>
<td>2.675</td>
<td>6.715</td>
<td>151.0</td>
</tr>
<tr>
<td>pl</td>
<td>1.491</td>
<td>2.588</td>
<td>73.6</td>
</tr>
<tr>
<td>vo</td>
<td>10.519</td>
<td>19.562</td>
<td>86.0</td>
</tr>
</tbody>
</table>

Table 4.6: Comparison of SA and Belady’s Algorithm

that is to be referenced the furthest in the future. An important measurement of the success of a caching algorithm is how well it compares to the optimal cache. Table 4.6 shows a comparison of the SA placement algorithm with the OPT (Belady) cache. Note that the Belady algorithm only applies when a cacheline contains only one instruction or if the cachelines are always contiguously executed. If there are multiple instructions per cacheline which execute at different times, then it may not be clear which line is “furthest in future”. Therefore, we have changed the SA algorithm to operate on lines of 1 instruction, so a chunk of size 256 bytes consists of 64 lines.

Thus, there is a significant gap between the SA placement and Belady’s algorithm ranging from 33.8% to 151% with an average gap of 95%. This is somewhat surprising since we saw above that SA is quite close to LRU and LRU long has been used as an approximate measure of an optimal cache. The main difference between LRU and Belady’s algorithm is that the LRU replacement merely tries to guess the future based on the past. Belady’s OPT cache, however, has full knowledge of the future. This results in a significant improvement. It may be that the cache size of 8KB is too small to exploit the full benefit of LRU. Therefore we plot the values of LRU and Belady for multiple values of the cache size in the range 1KB to 512KB. The results are shown in Figure 4-7.

2Recent work by Temam [14] extended Belady’s algorithm to cache lines of size larger than one, but only for a fixed description of the cache lines. Since code layout changes which instructions are grouped together into cache lines, this is still not a true optimum.
From these graphs we see that LRU does not in general converge to Belady until the cache is so large that the miss rate approaches zero (i.e. the only misses are cold start misses). This suggests that Belady’s OPT is strictly superior to LRU for any case in which the cache is not so big that it holds the whole executable.

4.8.3 Summary of Findings

We have now isolated three important factors for cache placement algorithms. For caches with more than one instruction per cache line, an extremely important factor is the spatial locality of the code. We achieved this through fluff separation and saw major improvements to all cache placement algorithms. The effect of code locality varies dramatically from benchmark to benchmark. The improvement ranges from nearly nothing to 70 percent or more depending on the application. A second criterion is associativity. This is found in both the LRU and Belady’s algorithm. However, associativity alone does not yield good placements as we saw in the comparison of the original LRU with the SA placement. Associativity combined with locality is needed to beat SA. The third factor is knowledge of the future. Belady’s algorithm is by far and away the best due to its perfect knowledge of the future. SA has only an approximate knowledge of the future contained in the cost metric. Even so, this is enough to give results on direct mapped caches approaching those from associative caches.

4.8.4 Hardness Results for Direct Mapped Code Layout

A natural question to ask is whether there exists an efficient algorithm for finding an optimal direct mapped cache. In other words we want to find an analog to Belady’s algorithm for direct mapped caches. Here, we show that there is no efficient algorithm unless P = NP.

The reduction is from the k-COLORING problem. Given an instance of k-COLORING we would like to reduce it to solving the direct mapped optimization problem for a particular trace and cache. So, given an undirected, unweighted graph
with \( n \) vertices and \( m \) edges, we wish to decide if it is \( k \)-colorable. Label the vertices 1, 2, \ldots, \( n \). Now for each vertex \( i \) we create a code block of size exactly one cache line. Furthermore, for each edge \((i, j)\) in the graph, we define a portion of the trace: \( b_i, b_j, b_i, b_j \). Finally, create new blocks \( f_{ij}^l \) for \( l = 1, 2, 3, \ldots k \) which are also one cache line long. Now place the sequence \( f_{ij}^1, f_{ij}^2, \ldots f_{ij}^k \) repeated 3 times after the edge sequence for \((i, j)\). Now suppose we want to map these blocks for a \( k \) line cache. Each sequence \( f_{ij} \) completely flushes the cache in the optimal layout, since if two of these \( k \) blocks map to the same line, we get at least two extra misses in the sequence, however we save at most one miss through preserving one of the old cache lines. This results in \( mk \) cache misses. In the edge sequences themselves, we have exactly 2 misses if \( b_i \) and \( b_j \) map to different cache lines and exactly 4 misses if they map to the same cache line. The total number of misses is therefore between \( m(k + 2) \) and \( m(k + 4) \). We see that the original graph is \( k \)-colorable if and only if there is some layout of the direct mapped cache in which every edge results in exactly two misses. Thus the graph is \( k \)-colorable if and only if the cache can be mapped with exactly \( m(k + 2) \) misses. So an optimal cache mapping results in a decision procedure for \( k \)-COLORING (and indeed an actual solution if one exists). Since \( k \)-COLORING is NP-hard, so is code layout.

So, there is no hope of finding the optimal cache miss rate for a direct mapped cache. We know that Belady’s algorithm serves as a lower bound for the optimal rate. However, we do not know how tight this lower bound is. A consequence of the proof above is that it is NP-hard to determine whether the Belady bound is achievable for a given direct mapped cache. The cache constructed in the proof always has \( m(k + 2) \) misses for Belady’s algorithm (The furthest in future algorithm implies that the blocks \( f_{ij}^l \) do not replace each other and hence flush the cache). So, Belady is an achievable bound if and only if the graph is \( k \)-colorable.
4.9 Summary

In summary, we have constructed a heuristic algorithm for optimizing direct mapped caches based on local search techniques. The new algorithm is asymptotically faster than the previously known greedy algorithms and gives just as good results. In fact, we found that if this algorithm is combined with fluff separation it does nearly as well as a fully associative cache with LRU replacement. We also compared our cache with the optimal fully associative cache derived from Belady’s algorithm. There is a significant gap between the two algorithms, but we show that not only is the optimal direct mapped cache NP-hard to find, but it is also NP-hard to determine if the Belady bound is achievable. Also, our results suggest that it is very hard to significantly improve upon the results shown here due to the flatness of the solution space.

4.10 Future Work

There are a number of open questions that remain to be answered. We have shown code layout for direct mapped caches is NP-Hard. Is it also NP-Hard to approximate? We conjecture that this is true based on its relationship with k-COLORING and, as we shall see in the next chapter, other NP-hard optimization problems which have been shown to be hard to approximate. A related question is, can one determine whether Belady’s algorithm is a tight lower bound to within $1 + \epsilon$ or more generally any constant fraction. Another line of research has to do with combining Belady’s algorithm with large cache lines. As it stands Belady’s algorithm only applies to cache with line size 1. Temam [14] extended this to an algorithm which determines the optimal associative cache with larger cachelines with a fixed layout. Young et al. [10] showed that maximizing spatial locality is reducible to the Traveling Salesman Problem. Perhaps a similar reduction can be done for Belady with an arbitrary layout of the code.
Figure 4-5: Effects of Varying Temperature Ratio on Simulated Annealing at Numerous Search Length
Figure 4-6: Effects of Simulation Length on Simulated Annealing
Figure 4-7: Comparison of LRU to Belady's OPT Cache
Chapter 5

Reductions to Combinatorial problems

The code layout problem using a pairwise heuristic metric can be written as follows:

\[ \min(\sum_{ij} c_{ij}o_{ij}) \]

Where \( i \) and \( j \) range over the code blocks, \( c_{ij} \) is the edge between \( i \) and \( j \) in the TRG, and \( o_{ij} \) is the amount of overlapping cache lines between blocks \( i \) and \( j \). In this section we reduce this problem to two other well-studied combinatorial optimization problems. The hope is that an algorithm which works well for the new problems will also work well for the original problem. Another goal is to find theoretical lower bounds for the conflict metric which can be used to tell how far away a placement is from optimum as well as potentially being used for branch-and-bound style exhaustive search algorithms.
5.1 Quadratic Assignment problem

5.1.1 Formulation

The general code layout problem using a pairwise metric as above can also be formulated as follows. Consider \( n \) potentially variable size code blocks. If each of these blocks is to be placed in one of \( l \) offset positions in a cache, then we can define a 0-1 variable \( x_{ip} \) to be 1 when block \( i \) is placed at location \( p \) and 0 otherwise. Since each block must be placed at exactly one location we have \( \sum_p x_{ip} = 1 \). Now for every pair of blocks \( i, j \) and possible locations \( p, q \), we associate a cost \( c_{ipjq} \) which is 0 if \( i \) and \( j \) do not overlap in cache under this placement, otherwise it just the value of the pairwise metric edge between \( i \) and \( j \) multiplied by the amount of overlap in cache. Now finding an optimal placement with respect to this metric is equivalent to solving the following 0-1 integer program:

Given \( n^2 l^2 \) cost coefficients \( c_{ipjq} \) \( (i, j = 1, 2, \ldots, n, p, q = 1, 2, \ldots, l) \) determine an \( n \) by \( l \) solution matrix \( X = (x_{ij}) \) so as to

\[
\begin{align*}
\text{Minimize} & \quad \sum_{ip} \sum_{jq} c_{ipjq} x_{ip} x_{jq} \\
\text{subject to} & \quad \sum_p x_{ip} = 1 \\
& \quad \text{and } x_{ip} = 0 \text{ or } 1.
\end{align*}
\]

This formulation of the code layout problem is related to a classic combinatorial optimization problem called the Quadratic Assignment problem. [15, 16, 17, 18]. The standard formulation of the Quadratic Assignment problem differs somewhat from above in that typically \( n = l \) and there is the additional constraint \( \sum_i x_{ip} = 1 \). The standard application is to minimize the cost of communication between \( n \) objects while placing each object at one of \( n \) distinct locations with exactly one object per location. However, the basic structure of the problem remains the same, so it may be hoped that techniques used to solve Quadratic Assignment will also apply to code layout.
5.1.2 Hardness

Unfortunately the Quadratic Assignment Problem is NP-hard as was shown by Sahni and Gonzalez [19]. Furthermore, they showed that the QAP is among the hardest of all NP-hard problems in that even finding an $\epsilon-$approximate solution is NP-hard. While their results were for the original Quadratic Assignment problem, the statements are still true for the relaxed version above. As we shall see in the next section the MAX $k$-CUT problem turns out to be a special case of this IP, and MAX $k$-CUT has been shown to be NP-hard to approximate. The implications are that we can’t hope for any general algorithm that is guaranteed to give a near optimal solution. However, there may be heuristic algorithms which work well in most cases.

5.1.3 Algorithms

A number of algorithms have been proposed for the QAP. We describe two very different approaches here.

Exact Algorithms  Obviously, the best algorithm for QAP would find the true optimum solution. Algorithms which achieve this are called exact algorithms. However, as we just mentioned QAP is NP-hard and so an exact algorithm most likely requires exponential time. Exact algorithms try to use pruning techniques to reduce the size of the search tree to a manageable size. The most successful of these techniques for the general quadratic assignment problems is branch-and-bound. [17] During each intermediate stage of branch-and-bound there is some set of objects assigned to locations. The search tree can now be pruned by lower bounding the value a solution can have with this set of assignments. If the value of the lower bound is higher than the current best-found solution, then the current branch is pruned. Branch-and-bound algorithms require tight, fast to compute lower bounds. For the QAP with “random” coefficients there exist lower bounds which, at least for small problems, can approximate the actual best solutions. Unfortunately we show below that for the particular type of coefficient matrix that arise in code placement, a large class of these lower
bounds reduce to 0 which dims the prospects of running branch-and-bound. In any case, even for the general QAP, current branch and bound techniques are only feasible for problems of size around 15 and so only potentially useful for the absolute smallest code placement algorithms. Other exact algorithm techniques such as cutting plane [20] or dynamic programming [21] also cannot solve problems of even moderate length.

**Local Search** The most successful algorithms for the QAP, both in terms of quality of solution and the size of problems that can be solved, have been local search algorithms. Simulated Annealing in particular has been studied by, among others, Burkard [26], Connoly [27] and Laursen [28]. For a set of test problems they showed that simulated annealing finds solutions better than the previous best known in a reasonable amount of time. Their success with simulated annealing with the general QAP inspired our use of simulated annealing in the code layout algorithm of the previous chapter. Another local search algorithm that has been applied to QAP is called *taboo search*. Taboo search, like simulated annealing, performs local moves and tries to avoid local optima. Taboo search chooses the local change resulting in the best improvement to the objective function at each step. If the system is at a local optimum, the least negative move is chosen. To avoid cycles, a list of forbidden moves is maintained. However, if a forbidden move results in a solution better than any previously seen, it is allowed to be made. Taboo search for the QAP was investigated by Taillard [29] and Skorin-Kapov [30] with results comparable to Simulated Annealing.

### 5.1.4 Lower Bounds

Finding sharp lower bounds for a combinatorial optimization is useful for a number of reasons. First, as mentioned above, they are crucial to branch-and-bound algorithms. The tighter the lower bound, the more pruning can be done in the search, and the larger the size of problems that can be efficiently solved. Another use of lower bounds is as a benchmark with which to test the success of approximate algorithms. If the lower bound is tight, than algorithms that find solutions near the optimum, will
also find solutions near the lower bound. More importantly, if some algorithm finds solutions near the lower bound then the solutions must also be close to optimum. Many lower bounds for the QAP have been proposed. The earliest, and the benchmark to which all subsequent lower bounds are compared, was described independently by Gilmore [31] and Lawler [16]. The theory behind the Gilmore-Lawler and related bounds is that although quadratic assignment is hard, linear assignment can be solved exactly using a greedy approach. Once a particular object has been assigned to a position, the minimum cost between this object and all the rest can be formulated as a linear assignment. Thus a simple lower bound for the QAP is to find this minimal relative cost for each object and position, and then find an assignment of objects to positions such that the sum of these relative costs is minimized. Formally, the Gilmore-Lawler bound is obtained by solving the linear assignment:

\[
GLB = \min\left(\sum_{jq} d_{jq} x_{jq}\right)
\]

where each \( d_{jq} \) is the solution to a linear assignment:

\[
d_{jq} = \min\left(\sum_{ip} c_{ipjq} x_{ip}\right)
\]

This bound works reasonable well for general QAP's. In fact no other proposed lower bound beats it by any significant amount consistently in practice. Unfortunately, it is easy to see that the Gilmore-Lawler bound (GLB) is always 0 when we are dealing with code placement. In the following arguments we assume that no single code block has size bigger than the cache, and also that every pair of blocks has sum of sizes at most as big as the cache. If these conditions are ever violated we can get trivial additions to our lower bound. I.e. for each block larger than the cache, add in the cost of its self overlap in cache. For each pair of blocks with sum of sizes bigger than the cache, these blocks must overlap in cache, so we can add to the lower bound the minimum cost of placing just these two blocks in cache. Ignoring these rare cases, we see that every \( d_{jq} \) is 0, since for each \( i \) choose \( p_{jq} \) such that placing \( i \) at \( p_{jq} \) and \( j \) at \( q \) results in no overlap. Now \( x_{ip} \) is always 0 unless \( p = p_{jq} \), but in this case \( c_{ipjq} = 0 \). Thus, with this assignment of the \( x_{ip} \)'s, the value of \( d_{jq} \) is 0.
Many lower bounds for the QAP are obtained by modifying the problem while maintaining the same optimal solution, and then applying the GLB to the modified problem. Frieze and Yadegar [15] propose the following modification scheme (changed somewhat to fit the slightly different form of QAP we are considering here): Extend the quadratic assignment to allow for a linear term with coefficients \( b_{ip} \). Thus we are now minimizing:

\[
\sum_{ip} \sum_{jq} c_{ipjq} x_{ip} x_{jq} + \sum_{ip} b_{ip} x_{ip}
\]

subject to the same constraints as before.

Of course each, \( b_{ip} \) is 0 in the original formulation. Frieze and Yadegar attempted to move as much information as possible from the QAP into the linear portion of the problem. Thus the problem might become closer to a pure linear assignment and GLB will give a good results. In particular they considered transformations in which you substitute \( \tilde{c}_{ipjq} = c_{ipjq} - \gamma_{ipj} \). Now, with the substitution \( \tilde{b}_{ip} = b_{ip} + \sum_j \gamma_{ipj} \) we get an equivalent problem:

\[
\text{minimize} \sum_{ip} \sum_{jq} \tilde{c}_{ipjq} x_{ip} x_{jq} + \sum_{ip} \tilde{b}_{ip} x_{ip}
\]

With the same minimum and solution as the original. We can now apply GLB to the modified problem to get:

\[
GLB(\gamma) = \min(\sum_{ip} (d_{ip} + \tilde{b}_{ip}) x_{ip})
\]

where

\[
d_{ip} = \min(\sum_{jq} \tilde{c}_{ipjq} x_{jq})
\]

Frieze and Yadegar examine these modifications and discover a method involving Lagrange multipliers for finding the optimum reduction of this form to give the best
possible bound. Unfortunately a similar argument as before shows that, under the same assumptions, no matter what the values of $\gamma$ are the modified bound is still always 0 for a code layout problem. Once again for each $d_{ip}$, we can choose an assignment where block $j$ is assigned to location $q_{ip}$ such that $c_{ipjq} = 0$ and thus $c_{ipj_{qip}} = -\gamma_{ipj}$. This gives us $d_{ip} \leq -\sum_j \gamma_{ipj}$. However, $\tilde{b}_{ip} = \sum_j \gamma_{ipj}$ and so $d_{ip} + \tilde{b}_{ip} \leq 0$. Thus the lower bound is at most a sum of non-positive terms, and thus at most 0. However, the lower bound is certainly at least 0, and so once again we get a bound of 0.

The problem we are encountering with these “nested linear assignment” techniques, is that there are too many zero coefficients in our cost matrix. In particular for every $i,p,j$ we can choose $q$ such that $c_{ipjq} = 0$. QAP’s with such sparse coefficients appear to be the hardest to lower bound. Perhaps there are other lower bound techniques which work for sparse cost matrices but we were not able to find any such in the literature.

**Relaxation Techniques** One of the difficulties in solving the QAP exactly is that the constraints are very tight. In particular the values $x_{ip}$ were required to be 0-1 variable satisfying certain sums. Many attempts at finding lower bounds proceed by relaxing these constraints to a form that is easier to solve. The GLB bound is a relaxation in which the quadratic term $x_{ip}x_{jq}$ is relaxed to a form $x_{ip}y_{jq}$ where the $x$’s and $y$’s are chosen independently allowing us to apply iterated linear assignment.

Another approach might be to relax the integrality of the variables $x_{ip}$. For example if we let them be real valued variables between 0 and 1, the solution to this relaxed problem is certainly a lower bound to the more constrained original problem. Recently, there has been a good deal of effort on certain types of relaxations of combinatorial optimization problems in which the constraints (or solutions) are positive semidefinite. The form of a semidefinite program is described in the next section.

Semidefinite relaxations for an important special case of the Quadratic Assignment Problem have been proposed by Zhao, Karisch, Rendl, and Wolkowicz [24],[25].
While, their approaches do not generalize to the form needed for code layout it is certainly conceivable that such a relaxation exists in this case as well.

5.1.5 Summary

We were able to write the code layout problem with a pairwise metric heuristic as an instance of a variant of the classical Quadratic Assignment problem. The fact that local search, and in particular simulated annealing, were the best algorithms for the general QAP led us to apply this techniques to the code layout problem. Efforts to find a good lower bound for the QAP formulation of code layout have been unsuccessful so far because of the problem of the “sparse” cost matrix.

5.2 MAX k-CUT problem

5.2.1 Formulation

We now consider the special case when all the code blocks are the same size. Recall that our chunk placement algorithm from the last chapter satisfies this property. In Chapter 4 we showed that if all blocks are the same size and the block size divides the size of the cache, then we can assume that all the blocks either overlap completely or not at all. Thus, if we have \( n \) blocks of size \( l \) with a pairwise cost metric \( C \) and a cache of size \( L \), the code layout problem become:

Partition the set \( \{1, 2, \ldots, n\} \) into \( k = \frac{L}{l} \) sets \( P_1, P_2, \ldots, P_k \) such that the following sum is minimized:

\[
\sum_{i=1}^{k} \sum_{\{r, s \in P_i\}} C_{rs}
\]

or alternatively maximize

\[
\sum_{1 \leq i < j \leq k} \sum_{r \in P_i \ s \in P_j} C_{rs}
\]

The second formulation follow from the first since instead of minimizing the edges involved in chunks mapping to the same set, we maximize the edges involving chunks that do not map to the same set. So the problem is to partition the vertices of a graph
into $k$ sets such that value of the cut edges, i.e the edges that cross the partition, is maximized. This problem is known as the MAX $k$-CUT problem and is another well-studied combinatorial optimization problem. Note that we are actually interested in the equivalent MIN $k$-UNCUT also known as MIN $k$-PARTITION problem, but, as we shall see, there are advantages to working with max cuts instead of min uncuts.

5.2.2 Hardness

It is not surprising that the MAX $k$-CUT problem is NP-hard. The natural follow up question is whether it can be approximated efficiently. First of all it can be shown that a random partition of the vertices into $k$ sets will cut an expected $1 - \frac{1}{k}$ of the total edge weight on the graph, and thus at least $1 - \frac{1}{k}$ of the optimal MAX $k$-CUT. This method can also be derandomized to get a deterministic algorithm (see [32] for example). Thus we can easily get a relative error of $\frac{1}{k}$. Furthermore, Kann et al. [33] prove that no approximation algorithm can do better than a $\frac{1}{34k}$ relative error.

So how does this impact the MIN $k$-UNCUT problem? Note that any approximate solution to MAX $k$-CUT is a lower bound on the true optimal MAX $k$-CUT. An upper bound on the optimal MAX $k$-CUT is the total edge weight on the graph. The corresponding MIN $k$-UNCUT solution is an upper bound on the actual solution, however the only lower bound is 0. So, no matter how close the approximamte MAX $k$-CUT solution is to the optimal, the corresponding approximate MIN $k$-UNCUT solution may still have an unbounded relative error from the true optimum. Even worse, Kann et al. [33] prove that it is NP-hard to find an approximate solution to MIN $k$-UNCUT on $n$ vertices to within even a factor of $n^{2-\epsilon}$! All of this looks very dire for our hopes of finding a good solution to MIN $k$-UNCUT, however, just as in the case of QAP, there may be heuristics that find a good solution in practice.

5.2.3 Algorithms

MAX $k$-CUT, like the QAP is very difficult to solve exactly for any reasonable sized values of the number of vertices. However, local search has proven to be very successful
in graph cut problems. A local search heuristic by Kernighan and Lin [37] works well in practice. Simulated annealing was applied with success to MAX CUT by Johnson et al [38] and Berry and Goldberg [39]. The simulated annealing code used for the code layout in the last chapter is based on an implementation by Berry for MAX CUT.

**Semidefinite Programming** A recent development in finding approximate solutions to combinatorial optimization problems is the use of semidefinite programming relaxations. A semidefinite program is an extension of linear programs where the constraints matrices are allowed to be arbitrary positive semidefinite, i.e., the eigenvalues are all non-negative. The form of a positive semidefinite program (SDP) is [22]

For all \( \bar{x} \in \mathbb{R}^n \):

\[
\text{minimize (or maximize) } c^T \bar{x}
\]

such that \( F(\bar{x}) \) is positive semidefinite

where

\[
F(\bar{x}) = F_0 + \sum_{i=1}^n x_i F_i
\]

with each \( F_i \in \mathbb{R}^{n \times n} \)

Since the variables here are real values and not integer or 0-1, the semidefinite program must be a relaxation of the original discrete optimization problem. An actual candidate solution would require somehow rounding the real variables to integer values, thus changing the value of the solution. However, the SDP solution itself is optimal for the real valued solutions, and thus in particular a lower bound (or upper bound in the case of maximization problems) for the true discrete optimization problem. Furthermore, a tight relaxation combined with a good rounding technique can result in solutions where the integer (rounded) and fractional (original SDP) solutions are close together. As the true optimum must lie in between these two values, the rounded solution must therefore be close to the true optimum. Semidefinite programs can be solved to any degree of accuracy in polynomial time using a search
technique called the interior point method developed by Karmakar for linear programs and extended to semidefinite programs by Nesterov and Nemerovsky[23].

Semidefinite programming as applied to max cut style problems has seen a number of advances in recent years. A ground breaking paper by Goemans and Williamson [34] applied semidefinite programming to the MAX-CUT problem (partition into 2 pieces). The main idea is to replace the 0-1 variables by vectors and use dot products instead of multiplication. Their algorithm resulted in a provable 0.87856 approximation algorithm, considerably better than the best known 0.75 approximation algorithm and the first substantial progress in MAX-CUT in twenty years. The algorithm was extended to a semidefinite relaxation of the MAX k-CUT problem by Frieze and Jerrum [35], which seems quite promising but only increased the provable relative error from $\frac{1}{k}$ to $\frac{1}{k} - 2 \log kk^{-2}$. However, the Goemans and Williamson algorithm for MAX-CUT, while provably only 0.878 of optimum, in practice gave results typically at least 0.96 of optimum. The Frieze and Jerrum approach may similarly do better in practice than what can be proven. Another advantage of semidefinite programming solutions is that the solution to the SDP is an upper bound for the MAX k-CUT problem. This results in the much sought after lower bound for the MIN k-UNCUT problem. While, in theory this lower bound cannot be guaranteed to be more than $\frac{1}{\pi^2}$ of the actual rounded solution, in practice it may be much closer.

We had hoped to test these hypotheses with an actual implementation of the Frieze and Jerrum algorithm. Unfortunately the space requirements, while polynomial, proved to be too much for our Pentium Pro with 256MB of RAM. However, a recent paper by Klein and Lu [36] succeeded in sufficiently reducing the space requirements for the original Goemans and Williamson MAX CUT algorithm. It may be that a similar approach will succeed in reducing the space requirements of MAX k-CUT to manageable levels. There appears to be some hope of this as the actual matrix of constraints from the Frieze Jerrum SDP is rather sparse. If such a method can be made practical than it would also serve as an alternate algorithm for computing a code layout.
5.2.4 Summary

This section showed how the fixed size code layout problem could be reduced to the MIN k-UNCUT or equivalently MAX k-CUT problem. We explored some of the difficulties of solving this problem, but suggest that it may be tractable using semidefinite programming. We also mentioned that local search techniques and simulated annealing have been successful on related problems.

5.3 Related Problems

5.3.1 Traveling Salesman Problem

A related problem to code layout for instruction cache is code linearization. This is the problem of trying to reorder blocks such that execution is as linear as possible. Code linearization is useful for improving branch prediction and increasing spatial locality in the program. It also has an effect on code layouts for instruction cache for two reasons. First of all, it improves the uniformity of the blocks in the conflict metric, thereby improving the accuracy of the GBSC metric. The second effect is that if two basic blocks share the same cacheline in a placement, then it improves the cache miss rate if they are usually executed together in the program. Karger, Smith, Johnson, and Young [10] reduced a form of code linearization (focusing on branch alignment) to another well-known combinatorial optimization problem, the Traveling salesman problem. They also present an algorithm for approximating the TSP and apply it code placement. This suggests that there may be some benefit from applying the TSP based code linearization algorithm before running layout for instruction cache.

5.4 Conclusion

The above sections reduce the code layout problem to a number of well-studied combinatorial optimization problems. This means that techniques employed to solve these
problems should also help for code layout. Furthermore, future advances in approximation solutions for QAP or MAX k-CUT should lead to corresponding better code layout algorithms.
Chapter 6

Conclusion

6.1 Contributions of This Thesis

This thesis provided an in depth look at trace based code layout algorithms for reducing conflict misses in direct mapped caches. There were three major portions of this thesis: methods to choose a good conflict metric, introducing local search as a method to optimize this metric, and reductions to combinatorial problems.

The metric we chose for measuring the amount of cache conflict in a code layout was the pairwise metric proposed by Gloy, Blackwell, Smith, and Calder. In addition we proposed some minor modifications to this metric including a probabilistic heuristic. We further improved the metric accuracy by improving the uniformity of its code blocks. This was done through a method dubbed “Fluff Separation” which moved all rarely used code away from the popular code.

In Chapter 4 we introduced local search as a way to optimize this metric. Two algorithms were presented. Local Optimization simply looks for a local minimum to the conflict metric. Simulated Annealing adapts methods used for simulating a cooling solid to try to avoid local minima. We found that Simulated Annealing gives a real but small improvement over Local Optimization. Both local search algorithm run in time $O(n^2)$, a significant improvement over previous “greedy” algorithms which ran in cubic time. Local Search gives results on direct mapped caches as good or better than the current state of the art and also performs nearly as well as a fully associative
cache with LRU. There is still a large gap between our placement and the optimal fully associative cache, however we showed that not only is finding an optimal direct mapped cache NP hard, but it is also NP hard to know if the optimal associative cache result is achievable. In summary, good placement can do nearly as well as associativity. But a combination of associativity and perfect knowledge of the future does far better than either one alone. In practice, associativity is very expensive and future knowledge is never perfect, thus our algorithm is very appealing.

Finally, in Chapter 5 we reduce the problem of optimizing our metric to two well-known combinatorial optimization problems: the Quadratic Assignment Problem and the Max k-Cut problem. Local search techniques are among the best algorithms for both of these problems. Both of these problems are NP-hard to solve and to approximate, but there is still hope of finding relatively tight bounds in practice. Unfortunately for code layout style problems, the known QAP bounds reduce to 0, and although a semidefinite programming approach may give tight bounds on Max k-Cut the space requirements were too great for an efficient implementation. A third problem, Traveling Salesman Problem, has been shown to be related to optimizing spatial locality, which we showed affected the cache miss rate, and thus may be applicable in conjunction with code layout techniques.

6.2 Future Work

There are a number of open problems related to code layout which may be the subject of future research. First of all, is there any metric more accurate than the GBSC metric? For efficiency reasons a pairwise metric is preferred, so is there an optimal pairwise metric which somehow captures the maximum information from the program trace? Another interesting line of research is to develop a metric capturing both temporal and spatial locality information, utilizing the relative importance of each in reducing the execution time of programs.

Both local search and greedy algorithms are giving almost the same cache miss rates. This suggests that they are both near a minimum for direct mapped caches. Is
there a tight lower bound for optimal direct mapped cache miss rates? We gave one lower bound based on Belady’s algorithm that does not appear to be tight. However, it is an NP-hard question to determine if the Belady bound is achievable which seems to reduce the likelihood of finding tighter bounds. While we showed that code layout for direct mapped caching is NP hard, the approximation version is still open. Is it NP-hard to approximate the optimal cache miss rate within some fraction $e$?

In regards to Belady’s algorithm it is still open on how to extend it to caches with more than one instruction per line when we have the flexibility to change the layout and thus change which instructions are loaded together. An interesting problem would be to find an algorithm that optimizes associative caches for cache miss rate given the ability to modify spatial locality.

Another approach to lower bounding the code layout problem comes through the reductions to QAP and Max k-Cut. Is there a lower bound on the QAP which gives a non-zero result when used with the very sparse matrices found in code layout? A similar question can be asked about Max k-Cut. There is some hope that semidefinite programming may be applied to these problems, especially the latter given the formulation by Frieze and Jerrum. However the SDP relaxation must be both time and space efficient to solve. Finally, the code layout problem may also have a model which is equivalent to a more tractable problem to solve or approximate than either QAP or Max k-Cut.
Appendix A

Graphs
Figure A-1: Comparisons of Caching Algorithms
Bibliography


