Caches and Algorithms

by

Anthony G. LaMarca

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

1996

Approved by_____

(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree_____

Date_____

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 1490 Eisenhower Place, P.O. Box 975, Ann Arbor, MI 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Caches and Algorithms

by Anthony G. LaMarca

Chairperson of Supervisory Committee: Professor Richard E. Ladner
Department of Computer Science
and Engineering

This thesis investigates the design and analysis of algorithms in the presence of caching. Since the introduction of caches, miss penalties have been steadily increasing relative to cycle times and have grown to the point where good performance cannot be achieved without good cache performance. Unfortunately, many fundamental algorithms were developed without considering caching. Worse still, most new algorithms being written do not take cache performance into account. Despite the complexity that caching adds to the programming and performance models, cache miss penalties have grown to the point that algorithm designers can no longer ignore the interaction between caches and algorithms.

To show the importance of this paradigm shift, this thesis focuses on demonstrating the potential performance gains of cache-conscious design. Efficient implementations of classic searching and sorting algorithms are examined for inefficiencies in their memory behavior, and simple memory optimizations are applied to them. The performance results demonstrate that these memory optimizations significantly reduce cache misses and improve overall performance. Reductions in cache misses range from 40% to 90%, and although these reductions come with an increase in instruction count, they translate into execution time speedups of up to a factor of two.

Since cache-conscious algorithm design is uncommon, it is not surprising that there is a lack of analytical tools to help algorithm designers understand the memory behavior of algorithms. This thesis also investigates techniques for analyzing the cache performance of algorithms. To explore the feasibility of a purely analytical technique, this thesis introduces *collective analysis*, a framework within which cache performance can be predicted as a function of both cache and algorithm configuration.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am extremely grateful to my advisor Richard Ladner who has given me constant support and encouragement over the years. Richard taught me the value of focusing on fundamental problems, and to appreciate the beauty of proving that things are true. I am also extremely grateful to Ed Lazowska who gave me relentless support and from whom I learned the value of perspective. I also wish to thank the other members of my committee: John Zahorjan, Larry Snyder, Brian Bershad and Anna Karlin. Their comments and feedback have greatly improved the quality of my work.

Special thanks go to Frankye Jones who shepherded me through the maze of university bureaucracy on many, many occasions.

I wish to thank Mike Salisbury, Jeff Dean, and Leslie LaMarca for reading drafts of this thesis. Their feedback significantly improved this document. I am also grateful to Nikolas and Diana Korolog for their long distance support and encouragement.

I owe many thanks to AT&T for providing me with the wonderful fellowship that has supported me over the past four years.

Finally I would like to thank my office mates, all of whom I have come to consider good friends. Thanks to Ted Romer for the endless technical help, to Neal Lesh for the philosophy, to Jeff Dean for the solidarity, to Dave Grove for help with my lemmata, and to Mike Salisbury for everything. Now that I am leaving, hopefully the office will be quiet and they can get some work done.

Chapter 1

Introduction

Since the introduction of caches, main memory has continued to grow slower relative to processor cycle times. The time to service a cache miss to memory has grown from 6 cycles for the Vax 11/780 to 120 for the AlphaServer 8400 [Clark 83, Fenwick et al. 95]. Cache miss penalties have grown to the point where good overall performance cannot be achieved without good cache performance. Unfortunately, many fundamental algorithms were developed without considering caching. The result is that "optimal" algorithms often perform poorly due to cache behavior.

Even today, most new algorithms being written do not take cache performance into account. Caches were originally introduced as a way to speed up memory accesses and these caches were transparent to the programmer. As a result, very few programmers had knowledge of the caching mechanism and the effects their programming decisions had on cache performance. Even though cache performance is now a major factor in overall performance, programmers still have little understanding of how caching works. Bad cache performance is often attributed to "cache effects" and is dismissed as randomness in the execution rather than a factor that can be understood and controlled.

In this thesis, I investigate how caches affect the performance of algorithms. I design and optimize algorithms for good cache behavior as well as analyze algorithm performance in models that take caching into account.

1.1 Caches

In order to speed up memory accesses, small high speed memories called *caches* are placed between the processor and the main memory. Accessing the cache is typically much faster than accessing main memory. Unfortunately, since caches are smaller than main memory they can hold only a subset of its contents. Memory accesses first consult the cache to see if it contains the desired data. If the data is found in the cache, the main memory need not be consulted and the access is considered to be a cache *hit*. If the data is not in the cache it is considered a *miss*, and the data must be loaded from main memory. On a miss, the block containing the accessed data is loaded into the cache in the hope that it will be used again in the future. The *hit ratio* is a measure of cache performance and is the total number of hits divided by the total number of accesses.

The major design parameters of caches are:

- *Capacity* which is the total number of bytes that the cache can hold.

- *Block size* which is the number of bytes that are loaded from and written to memory at a time.

- *Associativity* which indicates the number of different locations in the cache where a particular block can be loaded. In an *N-way set-associative* cache, a particular block can be loaded in N different cache locations. *Direct-mapped* caches have an associativity of one, and can load a particular block only in a single location. *Fully associative* caches are at the other extreme and can load blocks anywhere in the cache.

In most modern machines, more than one cache is placed between the processor and main memory. These hierarchies of caches are configured with the smallest, fastest cache next to the processor and the largest, slowest cache next to main memory. The smallest cache is typically on the same chip as the processor and accessing its data takes one or two cycles. These first level caches may be direct-mapped or set associative. The remaining caches are usually off chip and are slower to access than the level one cache but still much faster than main memory. These lower level caches are typically direct-mapped.

High cache hit ratios depend on a program's stream of memory references exhibiting locality. A program exhibits *temporal locality* if there is a good chance that an accessed data item will be accessed again in the near future. A program exhibits *spatial locality* if there is good chance that subsequently accessed data items are located closely together in memory. Most programs tend to exhibit both kinds of locality and typical hit ratios are greater than 90% [Lebeck & Wood 94].

1.2 Methodology

In this section I present the methodology I use in the design and analysis of algorithms and contrast it with approaches that have been used in the past.

1.2.1 Designing for Performance

Even when an algorithm is designed with performance in mind, its memory behavior is seldom considered. Typical design and optimization techniques only attempt to reduce the number of instructions that are executed. In this thesis, I investigate the benefits of designs which optimize cache performance as well as instruction count. The cache performance optimizations presented in this thesis focus exclusively on eliminating misses in the data reference stream and do not attempt to eliminate misses in the instruction stream. I do not attempt to reduce instruction cache misses as instruction cache behavior is usually quite good and there are effective compiler optimizations for improving instruction cache performance when it is not good [Hwu & Chang 89].

Cache locality is a good thing. When spatial and temporal locality can be improved at no cost it should always be done. In this thesis, however, I consider improving locality even when it results in an increase in the total number of executed instructions. This represents a significant departure from traditional design and optimization methodology. I take this approach in order to show how large an impact cache performance can have on overall performance. In Chapter 6 for instance, I show how restructuring iterative mergesort can increase the number of instructions executed by 70%, yet still produce an algorithm that sorts up to 75% faster due to a reduction in cache misses.

A drawback of designing algorithms for cache performance is that often none of the cache parameters are available to the programmer. This raises a dilemma. A

programmer might know that it would be more efficient to process the data in cache size blocks but cannot do so if the capacity of the cache is unknown. One approach used by some is to make a conservative assumption and rely on the cache to be some minimum size. I take a different approach, and assume that the exact cache parameters are exported to the programmer by the system. That is, I assume that the capacity, block size, associativity and miss penalty of the caches are known by the programmer.

This change clearly increases the complexity of the programmer's environment. Caches, which have traditionally been transparent to the programmer, are now exposed. This change also raises portability issues. While correctness is still preserved, codes compiled for one memory system may perform poorly if executed on a machine with a different memory system.

Despite these drawbacks, exporting cache parameters has the potential to greatly aid efficient algorithm design. In this thesis, I show how algorithms that are already efficient can be made to perform even better if specific architectural characteristics are known. I show how efficiency can be improved by making decisions at both compile time and at run time.

Programmers do not always need to think about cache behavior when writing code. Performance is a low priority for most code that is written. There are, however, bodies of code for which performance is an important factor and exporting the cache parameters offers a significant advantage in the development of efficient algorithms.

1.2.2 Analyzing Performance

The true test of performance is execution time, and numerous researchers investigate algorithm performance by comparing execution times. Unfortunately, execution times offer very little insight into the reasons for performance gains or losses. In Chapter 6, I propose a change to quicksort which increases instruction counts and reduces cache misses. While the resulting algorithm has very similar execution times to the standard quicksort, its performance tradeoffs are very different. While execution time is indeed the final measure of performance, as a metric it is too coarse-grained to use in the intermediate stages of analysis and optimization of algorithms.

The majority of researchers in the algorithm community compare algorithm performance using analyses in a unit-cost model. The RAM model [Aho & Ullman 83] is

used most commonly, and in this abstract architecture all basic operations, including reads and writes to memory, have unit cost. Unit-cost models have the advantage that they are simple to understand, easy to use and produce results that are easily compared. A serious drawback is that unit-cost models do not adequately represent the cost of memory hierarchies present in modern computers. In the past, they may have been fair indicators of performance, but that is no longer true.

It is also common for the analyses of algorithms in a specific area to only count particular expensive operations. Analyses of searching algorithms, for example, typically count only the number of comparisons performed. The motivation behind counting only expensive operations is a sound one. It allows the analyses to be simplified yet retain accuracy since the bulk of the costs are captured. The problem with this approach is that shifts in technology can render the expensive operations inexpensive and *vice versa*. Such is the case with the compare instructions performed in searching algorithms. On modern machines comparing two values is no more expensive than copying or adding them. In Chapter 3, I show how this type of analysis can lead to incorrect conclusions regarding performance.

In this thesis, I employ an incremental approach to evaluating algorithm performance. Rather than discard existing analyses and perform them again in a new comprehensive model, I leverage as much as possible off of existing analyses and augment them where they are weak. An incremental approach involves less work than a complete re-evaluation and offers an easy transition for those interested in a more accurate analysis.

A major weakness of unit-cost analysis is that it fails to measure the cost of cache misses that algorithms incur. For this reason, I divide total execution time into two parts. The first part is the time the algorithm would spend executing in a system where cache misses do not occur. I refer to this as the *instruction cost* of the algorithm. The second part is the time spent servicing the cache misses that do occur, and I call this the *memory overhead*.

I measure instruction cost in two ways: with analyses in a unit-cost model and with dynamic instruction counts from actual implementations. Neither unit-cost analyses or dynamic instruction counts will measure instruction cost perfectly. Execution delays such as branch delay stalls and TLB misses are not measured with either of these methods, nor do they capture variance in the cycle times of different types

of instructions. Despite these shortcomings they both provide a good indication of relative execution time ignoring cache misses.

I measure memory overhead using trace-driven cache simulation. Cache simulations have the benefit that they are easy to run and that the results are accurate. In Chapters 4 and 7, I also explore ways in which the memory overhead of an algorithm might be predicted without address traces or implementations. When both analytical predictions and simulation results are available, I compare them to validate the accuracy of the predictions.

All of the dynamic instruction counts and cache simulation results in this thesis were measured using Atom [Srivastava & Eustace 94]. Atom is a toolkit developed by DEC for instrumenting program executables on Alpha workstations. Dynamic instruction counts were obtained by inserting an increment to an instruction counter after each instruction executed by the algorithm. Cache performance was determined by inserting calls after every load and store to maintain the state of a simulated cache and to keep track of hit and miss statistics. In all cases, I configured the simulated cache's block size and capacity to be the same as the second level cache of the architecture used to measure execution time. The majority of the experiments in this dissertation were run on a DEC Alphastation 250. In Chapter 3 a number of other architectures are used to demonstrate the general applicability of my optimizations.

1.3 Thesis Contributions

This dissertation makes the following contributions:

- *An illustration of how architectural trends have affected the relative performance of algorithms*

  I revisit Jones's performance study of priority queues [Jones 86] and reproduce a subset of his experiments. The relative performance result I obtain on today's machine differ greatly from the machines of only ten years ago. In the original study, heaps performed only marginally and pointer-based self-balancing structures such as splay trees performed best overall. Due to high cache miss rates, however, the pointer based structures performed poorly in my experiments. The compact structure and memory-efficient behavior of implicit heaps gives them

a significant advantage and they outperformed all other priority queues in my evaluation.

- *A demonstration of the potential performance gains of cache-conscious algorithm design*

  I begin with efficient implementations of classic searching and sorting algorithms. I then show how applying simple memory optimizations can greatly reduce cache misses and improve overall performance. My simple techniques reduce cache misses by as much as 90% and speedup execution time by as much a factor of two. The majority of my studies are performed in a uniprocessor environment. In Chapter 7, I examine the impact that cache-conscious design has on lock-free synchronization in a shared-memory multiprocessor.

- *A set of optimizations that can be used to improve the cache performance of algorithms*

  These represent a set of optimizations that are both easy to apply and have a significant impact on performance. None of these optimizations is new. My contribution is a demonstration of how these techniques can be applied to real algorithms and evidence of the performance benefits they provide.

- *Algorithms and data structures with excellent performance*

  A byproduct of my study of cache-conscious design is a number of algorithms with excellent performance. Chapter 3 presents an implicit heap that performs add and remove operations up to twice as fast as traditional implicit heaps. Chapter 6 presents an improved mergesort which sorts up to 75% faster than a traditionally optimized iterative mergesort.

- *An analytical tool that can be used to predict the cache performance of algorithms*

  I develop *collective analysis*, a framework that allows cache performance to be analyzed as a function of both cache and algorithm configuration. Collective analysis is performed directly on the algorithm without implementations and address traces and predicts cache hit ratios for direct-mapped caches.

1.4 Overview

Chapter 2 surveys existing research that considers the effects that caches have on algorithm performance. The survey includes studies of particular algorithms as well as tools that enable cache-conscious design in general. It also identifies areas in which the effects of caching have not been sufficiently considered.

Chapter 3 presents an optimization of the memory system performance of heaps. I test the performance of both traditional heaps and memory optimized heaps using heapsort and by using them as a simulation event queue. In order to demonstrate the robustness of the optimizations, I measure the speedups the improved heaps provide for five different architectures with varying memory systems.

Chapter 4 introduces *collective analysis*, a framework that can be used to predict an algorithm's cache performance. I perform collective analysis on the heaps developed in Chapter 3, and its predictions are compared with the results of a trace-driven cache simulation.

Chapter 5 reproduces a subset of Jones's performance study of priority queues [Jones 86]. I compare the performance of implicit heaps, skew heaps and both top-down and bottom-up splay trees in the hold model. My work illustrates how shifts in technology can change the relative performance of algorithms. I point out the differences between my results and Jones's and explain why they occur.

Chapter 6 studies the cache performance of the heapsort, mergesort and quicksort algorithms. For each algorithm, I begin with implementations that have been heavily optimized using traditional techniques. I then show how the performance of all three algorithms can be improved by optimizing their cache behavior.

Chapter 7 examines the impact that caching has on the performance of lock-free synchronization in shared-memory multiprocessors. It presents a simple model of performance based only on the number of memory operations performed. It then shows that this simple model is a fair indicator of the relative performance of lock-free synchronization algorithms.

Finally, Chapter 8 offers concluding remarks.

Chapter 2

Caches and Algorithms

In this chapter, I examine previous work that considers the effects caches have on the performance of sequential algorithms. This work includes both studies of particular algorithms as well as tools and models that help facilitate cache-conscious design in general. Having reviewed previous work on this topic, I then identify important areas in which the effects of caching have not been considered sufficiently.

2.1 Successes

This section contains what I consider to be the successes in the area of cache-conscious algorithm design and analysis. I have divided the work into six rough categories: case studies of the cache performance of existing applications, compiler optimizations for improving locality, system support for improving cache performance, architectural models that account for caches, and external algorithms which can be converted to cache-efficient algorithms.

2.1.1 Case Studies

There are a number of case studies of the cache performance of specific applications. In most cases, the memory system behavior of the piece of code is investigated using profiling tools. Typically, this investigation leads to optimizations that speed up the application by a factor of two or more. Agarwal *et al.* optimized a library of linear algebra routines by tuning its memory behavior to the Power2 architecture [Agarwal et al. 94]. Martonosi *et al.* reorder data in order to optimize part of a sparse matrix equation solver [Martonosi et al. 95].

The vast majority of these studies focuses on codes with execution times that are dominated by loops over arrays. There are very few studies of the cache performance of applications that predominantly use pointer-based structures or that access arrays in irregular ways. One exception was a study by Koopman *et al.*, in which trace driven simulation was used to study the impact that different cache parameters have on the performance of combinator graph reduction [Koopman et al. 92].

2.1.2 Optimizing Compilers

There are a number of compiler optimizations for improving the cache locality of code. The framework for these optimizations was originally developed for parallelizing code, but it became clear that the same techniques could be used to improve the cache locality of code on a sequential machine. Changing the data layout of an array can greatly increase spatial locality [Cierniak & Li 95], interchanging loops has the potential to increase both spatial and temporal locality [Carr et al. 94], and loop tiling can increase temporal locality by deepening a loop nest and reducing the number of cache misses between data reuses [Wolfe 89]. Models have been developed that predict the locality of a set of loops and these models are used to guide the loop restructuring process [Kennedy & McKinley 92, Carr et al. 94, Gannon et al. 88]. These locality improving optimizations have been implemented in research systems including Stanford's SUIF compiler [Wolf & Lam 91] and the Memoria source-to-source translator [Carr et al. 94].

Using compiler optimizations to improve cache performance is clearly a worthwhile goal. If the difficult task of analyzing and improving cache performance can be hidden in the compiler, programmers are free to express programs in a natural way and not concern themselves with the associativity and block sizes of their caches. Unfortunately, existing compiler optimizations have very limited scope. With few exceptions these optimizations are restricted to loops over arrays, and most restrict the indices used to access the array. The result is that these optimizing compilers are targeted for scientific applications that tend to loop over arrays in very regular ways. Due to the difficultly of program analysis, however, these compilers offer little benefit to the myriad applications that use pointer-based structures and access arrays in irregular ways.

2.1.3 System Support

Most machines have physically indexed caches and support paged virtual memory. Many operating systems implement virtual to physical page mapping policies that attempt to minimize cache misses [Taylor et al. 90]. Careful page mapping policies have been shown to reduce cache misses by as much as 20% over random mappings [Kessler & Hill 92]. In the event that a bad mapping decision is made, Bershad *et al.* have developed techniques for detecting when heavily accessed pages conflict in the cache and for dynamically remapping them [Bershad et al. 94b].

Careful consideration has also been given to the effect that the runtime system has on cache performance. Grunwald *et al.* investigate the impact that memory allocation algorithms have on cache performance [Grunwald et al. 93]. They show that allocation algorithms that coalesce free blocks result in better locality that the traditional sequential first-fit algorithm. Similarly, there have been a number of studies investigating the impact that garbage collection algorithms have on cache performance [Diwan et al. 94].

2.1.4 Cache Performance Tools

There are a number of tools that can be used to investigate the cache performance of algorithms. These tools fall into different categories based on their strengths and weaknesses. In this section these categories are discussed and are evaluated based on the following criteria:

- *Usability* indicating how easy it is to apply the tool to algorithms to obtain performance predictions.
- *Accuracy* indicating how accurate the performance predictions of the tool are.
- *Intuition* indicating how much insight the tools provide as to why the algorithm performs the way it does.
- *Restrictions* identifying any limitations on the class of algorithms that the tool can be applied to.

A summary of the evaluation of the categories of tools according to these criteria is shown in Table 2.1.

Table 2.1: A comparison of cache-performance tools.

Techniques	*Usability*	*Accuracy*	*Intuition*	*Restrictions*
Hardware monitoring and trace-driven simulation	easy	high	low	-
Enhanced trace-driven	easy	high	medium	-
Hybrid	easy	low-medium	low-medium	-
Pure analytical	hard	med-high	high	oblivious

Hardware monitoring of a machine is a fast and accurate way to determine cache performance. Some architectures provide support for this in the form of counters that record the number of accesses, the number of cache misses and other performance metrics [Welbon et al. 94, Dig 92]. To allow more detailed monitoring, Horowitz *et al.* propose changes to the architecture that allow programs to execute special case code when cache misses occur [Horowitz et al. 96]. In the absence of such architectural support, expensive hardware can be used to monitor the cache behavior of a machine [Uhlig et al. 94]. The disadvantage of hardware monitoring, aside from potential cost, is that it usually can provide results only for an existing hardware configuration.

Most analysis of cache performance is currently done with trace-driven simulation [Mattson et al. 70, Smith 85]. Since the cache is simulated, this technique offers the flexibility to investigate arbitrary cache configurations, and data structures have been developed to allow a number of cache configurations to be simulated simultaneously [Hill & Smith 89]. The only real drawback of trace-driven simulation is that running the simulations can be time consuming.

Martonosi *et al.* augment trace-driven cache simulation by categorizing the cache misses by the type of miss and by the name of the data structure incurring the miss [Martonosi et al. 95]. The Memspy system they developed allows the programmer to see a breakdown of cache misses for each data structure. Tools like this are ideal for a programmer interested in optimizing an application's memory performance. In Table 2.1, I label these the enhanced trace-driven techniques. Systems with similar features include include Cprof [Lebeck & Wood 94] and SHMAP [Dongarra et al. 90].

Neither hardware monitoring nor trace-driven simulation offer the benefits of an analytical cache model, namely the ability to quickly obtain estimates of cache performance for varying configurations. An analytical cache model also has the inherent advantage that it helps a designer understand the algorithm and helps point out possible optimizations. A number of researchers have tried, with varying degrees of success, to combine trace-driven cache simulation with an analytical performance model. These techniques process an address trace and reduce it to a few key parameters describing an application's behavior. These parameters are then used as input to an analytical cache model from which predictions are made. In Table 2.1, I label these as hybrid techniques. Singh *et al.* compress an address trace into four quantities representing the working set size, the spatial locality, the temporal locality and the spatial-temporal interactivity of the application [Singh et al. 92]. These four quantities are then plugged into a formula to predict cache hit ratios. Unfortunately, either their quantities or their formula was poorly chosen and their predictions are inaccurate when the predicted cache configuration differs significantly from the measured configuration.

Agarwal *et al.* use a similar approach and reduce traces to quantities representing the rate of memory references, the rate of references to new cache blocks and the rate of cache misses [Agarwal et al. 89]. Unlike Singh's formula, which was chosen by curve fitting data, Agarwal's cache predictions come from a simple but sound cache model they derive. Given the small number of factors to which they reduce an address trace, their predictions match observed performance surprisingly well.

Finally, Temam *et al.* have developed an approach for quantifying the cache performance of algorithms using purely analytical techniques, and their method requires no implementations or address traces [Temam et al. 94]. In their model, the memory reference patterns are determined by studying the algorithm, and cache performance is predicted by determining when each data item is reused and how often this reuse is disrupted. Their model is similar but more complicated and accurate than the models automatically applied by current compilers. The main limitation of their work is that the algorithm to be analyzed is restricted to loop nests with oblivious reference patterns. No data dependent control flow or reference patterns are allowed. While the authors show good results for matrix multiply [Temam et al. 95], the restriction of an oblivious reference pattern limits the applicability of their model.

2.1.5 Architectural Models

The majority of algorithms have been designed and analyzed in an abstract architecture in which all operations have unit cost. While this may have been a reasonable model in the past, increasing cache-miss penalties have rendered this model a poor abstraction of the modern machine. Accordingly, a number of abstract architectures for sequential machines have been introduced that take into account the memory hierarchies present in modern machines.

The uniform memory hierarchy [Alpern et al. 94] models the memory system as a series of increasingly large memory modules. Computation can take place only in the smallest of these modules. Communication takes place via busses of varying bandwidth that connect neighboring modules. As well as the computation that is to occur, an algorithm describes when blocks of data should be moved between modules. The *efficiency* of a program in the uniform memory hierarchy is defined to be the percentage of time that computation occurs in the smallest module compared to the time spent waiting for a communication to finish.

The hierarchical memory model [Aggarwal et al. 87b] has a single address space in which an access to location i takes time $\lceil \log i \rceil$. Efficient programming in this model requires commonly used data items to be copied from high addresses to low addresses during their periods of high use. Upper and lower bounds for FFT and matrix multiply are given for the model. Aggarawal *et al.* also developed the hierarchical memory model with block transfer [Aggarwal et al. 87a]. The addition of a block transfer to their original hierarchical memory model is intended to capture the way physical memory incurs a high latency followed by high throughput.

Finally, others have used an even simpler model to determine the upper and lower bound for the communication complexity of various common tasks. The research assumes that a computation only has a space S in which to perform its work. A piece of memory may be brought in and out of this work space and the total number of these I/O operations is C. Tak *et al.* show that multiplying two n by n matrices exhibits the space communication tradeoff $CS = \Theta(n^3)$ [Lam et al. 89]. Aggarwal and Vitter prove lower bounds on the communication complexity of sorting and have developed variants of mergesort and distribution sort that are within a constant factor of optimal in this model [Aggarwal & Vitter 88].

2.1.6 External Algorithms

A final area that is relevant to cache-conscious design is the large body of algorithms optimized to minimize the number of disk accesses. Numerous papers study external sorting and searching algorithms [Knuth 73, Verkamo 88, Wegner & Teuhola 89, Verkamo 89]. Despite the difference in scale between pages and cache blocks, techniques used to reduce disk accesses can be used to reduce cache misses. Changing the word "page" to "cache block" in Gotlieb's paper on search trees [Gotlieb 81] produces a multi-way search tree that minimizes the number of cache blocks touched during an access.

One difficulty that can arise in the conversion of an external algorithm is that external algorithms assume control over the page replacement policy. With hardware caches, however, when misses occur the algorithm has no choice as to where that new block is loaded or which block is ejected. Thus external algorithms that depend on particular replacement policies for efficiency will likely not be of much use.

Another difficulty with this conversion is that disk accesses are far more expensive than cache misses. Because disk accesses are thousands of times more expensive than instructions, external algorithms can afford to greatly increase instruction cost to reduce the number of disk accesses. A straightforward adaption of an external technique may result in a low number of cache misses but bad overall performance due to high instruction cost.

2.2 Opportunities

The community that designs and analyzes practical algorithms has almost completely ignored caching. Considering the effect that caching has on algorithm performance, optimizing cache behavior offers one of the best opportunities for improving performance. At this point it is important to make a distinction between what I consider to be *practical algorithms* and *theoretical algorithms*. I call the algorithms found in programming textbooks and standard code libraries practical algorithms. They are algorithms that are developed for use in practice and are often tuned for performance. Theoretical algorithms, on the other hand, are designed with asymptotic performance in mind, and constant factors in their performance are irrelevant. These algorithms are intended as proofs of upper bounds on the complexity of problems. As

we saw in the previous section, researchers have produced cache-conscious theoretical algorithms for common problems. With very few exceptions, however, the design of practical algorithms has completely ignored caching.

A related opportunity lies in the area of characterizing the memory behavior of algorithms. A practitioner choosing between two algorithms can draw on a wealth of existing analyses to compare their expected instruction counts. Knuth, for instance, tells us that in the MIX model which has unit-cost memory references, mergesort executes at an average cost of $14.43n \log n$ verses $10.63n \log n$ for quicksort [Knuth 73]. Unfortunately, no such analyses exist for these algorithms in a model that takes caching into account.

Characterization of memory behavior is an important part of depicting how an algorithm performs. This characterization of memory system behavior might take the form of an analysis in a model that has as parameters cache size, block size and associativity. It might be as simple as a graph of cache miss ratios from a trace-driven simulation for a popular memory-system configuration. It is difficult to predict what level of detail the algorithm community will adopt, but these characterizations are needed so that informed decisions can be made regarding algorithm performance.

2.3 Summary

This chapter reviews existing work on caches and their effect on algorithm performance. It examines studies of the memory system performance of specific algorithms as well as compiler optimizations for improving the locality of algorithms in general. It also examines existing tools for analyzing the cache performance of algorithms and architectural models that account for caching.

The chapter identifies two important areas in which the effects of caching have not been seriously considered. First, the designers of algorithms for use in practice have paid little attention to caching. Due to high miss penalties, reducing cache misses represents an excellent opportunity for improving overall performance. The second opportunity lies in the characterization of the memory performance of algorithms. Existing analyses ignore caching and use models in which memory accesses have unit cost. A characterization of performance that includes cache overhead as well as instruction count will lead to a more accurate assessment of an algorithm's performance in practice.

Chapter 3

Optimizing Implicit Heaps

In this chapter, I examine and optimize the memory system performance of implicit heaps [Williams 64]. Heaps are a classic data-structure and are commonly used as event queues and for sorting. One of the goals of this thesis is to demonstrate the gains of cache-conscious design. Towards this goal, I develop two memory optimizations that can be applied to heaps and test the effects they have on performance using both cache simulations and actual executions. A byproduct of this performance study is a heap implementation with excellent overall performance.

3.1 Implicit Heaps

I begin by describing the implicit binary heap, and throughout this dissertation I refer to this as a *traditional heap*. The traditional heap is an array representation of a complete binary tree with N elements. All levels of the tree are completely full except for the lowest, which is filled from left to right. The tree has depth $\lceil \log_2(N+1) \rceil$. Each element i in the heap has a key value $Key[i]$ and optionally some associated data. The N heap elements are stored in array elements $0 \cdots N-1$; the root is in position 0, and the elements have the following relationships (see Figure 3.1)

$$Parent(i) = \lfloor \tfrac{i-1}{2} \rfloor, \text{ if } i > 0$$

$$LeftChild(i) = 2i + 1$$

$$RightChild(i) = 2i + 2$$

A heap must satisfy the *heap property*, which says that for all elements i except the root, $Key[Parent(i)] \leq Key[i]$. It follows that the minimum key in the data structure must be at the root.

In my performance evaluation, I use heaps as priority queues that provide the *add* and *remove-min* operations. In my implementation of *add*, the new element is added to the end of the array and is then percolated up the tree until the heap property is restored. In my *remove-min* implementation, the root is replaced with the last element in the array which is percolated down the tree by swapping it with the minimum of its children until the heap property is restored. I do not consider the implication nor the optimization of other priority queue operations such as *reduce-min* or the merging of two queues. Heaps have been well studied, and there are numerous extensions and more sophisticated algorithms for adding and removing elements [Knuth 73, De Graffe & Kosters 92, Carlsson 91, Gonnet & Munro 86]. In practice, however, few of these extensions improve the performance of heaps for the tasks I consider in this study.

3.2 A Motivating Example

Traditional algorithm design and analysis has for the most part ignored caches. One of the goals of this dissertation is to show that ignoring cache behavior can result in misleading conclusions regarding an algorithm's performance. In order to demonstrate this I present an examination of algorithms for building heaps.

Building a heap from a set of unordered keys is typically done using one of two algorithms. The first algorithm is the obvious and naive way, namely to start with an empty heap and repeatedly perform *add*s until the heap is built. I call this the

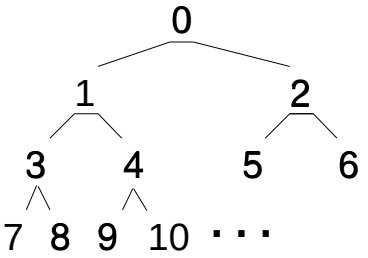

Figure 3.1: The structure of a binary heap.

Repeated-Adds algorithm.

The second algorithm for building a heap is due to Floyd [Floyd 64] and builds a heap in fewer instructions than Repeated-Adds. Floyd's method begins by treating the array of unordered keys as if it were a heap. It then starts half-way into the heap and re-heapifies subtrees from the middle up until the entire heap is valid. The general consensus is that due to its low instruction cost and linear worst case behavior Floyd's method is the preferred algorithm for building a heap from a set of keys [Sedgewick 88, Weiss 95, Cormen et al. 90].

To validate this, I executed both build-heap algorithms on a set of uniformly distributed keys varying in size from 8,000 to 4,096,000 elements. As the literature suggests, Floyd's method executes far fewer instructions per key than does Repeated-Adds. Runs on a DEC Alphastation 250 showed that for uniformly distributed keys, both algorithms executed a number of instructions linear in the number of elements in the heap. Floyd's algorithm executed 22 instructions per element on average while Repeated-Adds averaged 33 instructions per element. An evaluation based only on instruction cost thus indicates that Floyd's method is the algorithm of choice. When we consider cache performance, however, we see a very different picture.

First consider the locality of the Repeated-Adds algorithm. An *add* operation can only touch a chain of elements from the new node at the bottom up to the root. Given that an element has just been added, the expected number of uncached elements touched on the next *add* is likely to be very small (Figure 3.2). There is a 50% chance that the previously added element and the new element have the same

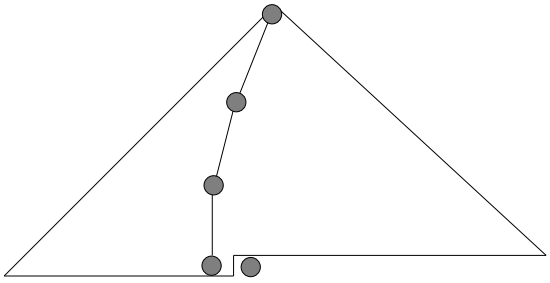

Figure 3.2: Building heaps with Repeated-Adds.

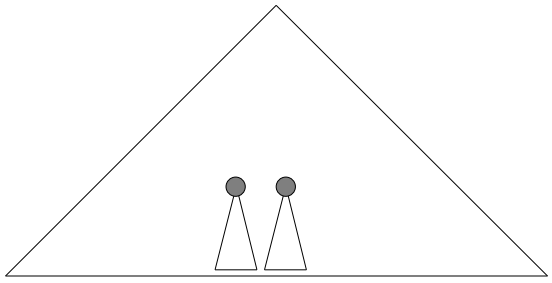

Figure 3.3: Building heaps with Floyd's method.

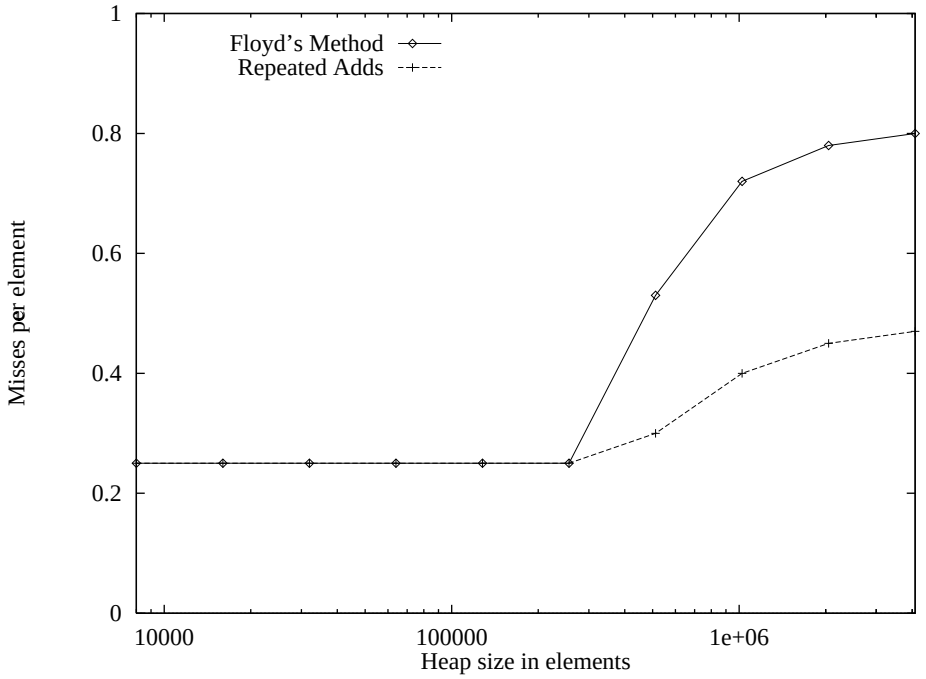

Figure 3.4: Cache performance of Repeated-Adds vs Floyd's method. Simulated cache size is 2 megabytes, block size is 32 bytes and heap element size is 8 bytes.

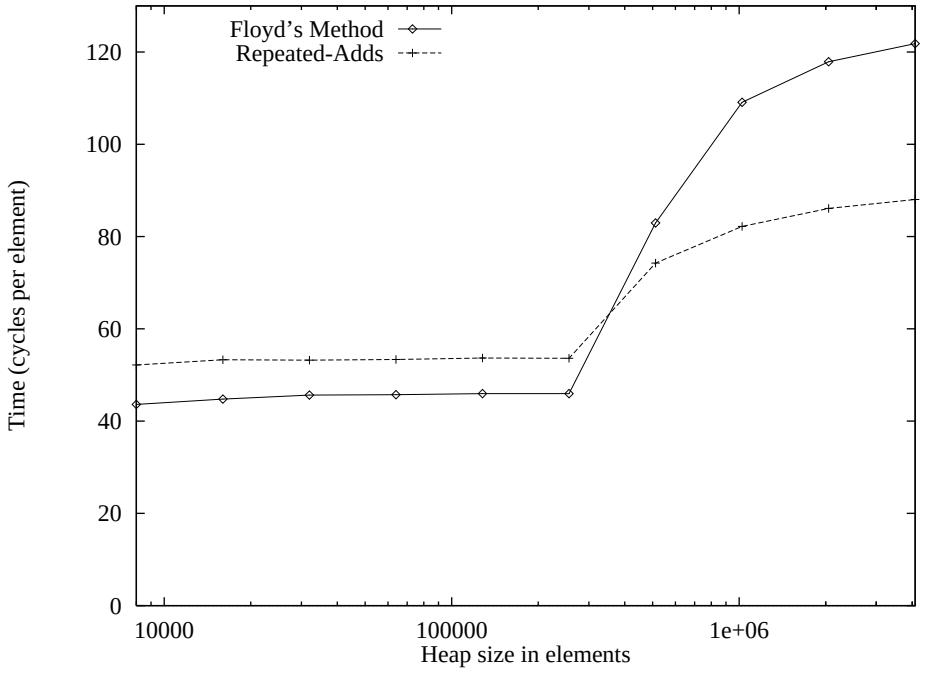

Figure 3.5: Execution times for Repeated-Adds vs Floyd's method. Executions were run on a DEC Alphastation 250 using 8 byte heap element.

parent, a 75% chance that they have the same grand-parent and so on. Thus, we expect the number of new heap elements that need to be brought into the cache for each *add* operation to be small on average.

Now consider the cache locality of Floyd's method. Recall that Floyd's method works its way up the heap, re-heapifying subtrees until it reaches the root. For $i > 1$, the subtree rooted at i does not share a single heap element with the subtree rooted at $i-1$ (Figure 3.3). Thus, as the algorithm progresses re-heapifying successive subtrees, it should exhibit poor temporal locality and incur a large number of cache misses.

To test this, I perform a trace driven cache simulation of these two algorithms for building a heap. Figure 3.4 shows a graph of the cache misses per element for a two megabyte direct-mapped cache with a 32 byte block size using 8 byte heap elements and varying the heap size from 8,000 to 4,096,000 elements. In this configuration, 2 megabytes / 8 bytes = 262,144 heap elements fit in the cache. This graph shows that up to this size, both algorithms only incur 0.25 cache misses per element. These represent the compulsory misses and since 4 heap elements fit per cache block, compulsory misses average out to 0.25 per element. Once the heap is larger than the cache, however, Floyd's method incurs significantly more misses per element than Repeated-Adds as our informal analysis suggested.

Figure 3.5 shows the execution times of these two algorithms on a DEC Alphastation 250 using 8 byte heap elements and varying the heap size from 8,000 to 4,096,000 elements. When the heap fits in the cache, both algorithms incur only compulsory misses and Floyd's method is the clear choice. When the heap is larger than the cache, however, the difference in cache misses outweighs the difference in instruction cost and the Repeated-Adds algorithm prevails.

This simple examination of algorithms for building heaps is strong motivation for analyses which account for the effects of caching. I examined seven algorithm textbooks and all of them recommend Floyd's method as the preferred technique for building heaps [Cormen et al. 90, Sedgewick 88, Weiss 95, Knuth 73, Manber 89, Aho & Ullman 83, Reingold & Wilfred 86]. If their authors had considered caching, as this example did, they would have been able to point out that due to poor locality, Floyd's method may perform worse than the Repeated-Adds algorithm for data sets that do not fit in the cache.

3.3 Optimizing *Remove-min*

This section examines the memory behavior of heaps and develops memory optimizations that increase their spatial and temporal reuse of data. In almost all cases, the proposed memory optimizations reduce memory overhead and either increase or do not change instruction cost. Optimizations are considered to be beneficial if the reduction in memory overhead outweighs the increase in instruction cost.

For a stream of operations on a heap where the number of adds and the number of removes are roughly equal, the work performed will be dominated by the cost of the removes. Doberkat [Doberkat 81] shows that independent of N, if the keys are uniformly distributed, *add* operations percolate the new item up only 1.6 levels on average. Doberkat [Doberkat 82] also studies the cost of the *remove-min* operation on a heap chosen at random from the set of legal heaps. He shows that after *remove-min* swaps the last element to the root, the swapped element is percolated down more than $(depth - 1)$ levels on average. Accordingly, I focus on reducing the memory overhead of the *remove-min* operation.

The first observation about the memory behavior of *remove-min* is that we do not want siblings to reside on different cache blocks. Recall that *remove-min* moves down the tree by finding the minimum of a pair of unsorted siblings and then swapping that child with the parent. Since both children need to be loaded into the cache, I would like to guarantee that both children are in the same cache block.

Figure 3.6 shows the way a heap will lay in the cache if four heap elements fit in a cache block and the heap starts at the beginning of a cache block. Unfortunately, in this configuration, half of the sets of siblings cross cache block boundaries. The result is that we will often need to bring two blocks into the cache despite the fact that we need less than one block of data. Lebeck and Wood refer to these as *alignment misses* [Lebeck & Wood 94]. Similar behavior occurs for other cache configurations where the cache block size is an even multiple of heap element size and the memory allocated for the heap starts a cache block.

There is a straightforward solution to this problem, namely to pad the array so that the heap does not start a cache block. Figure 3.7 shows this padding and its effect on the layout of the heap in the cache. This optimization will have its biggest impact when the number of heap elements per cache block is two. When two elements fit on a cache block, every pair of siblings will cross a cache block boundary with an

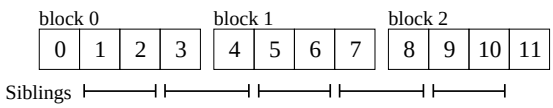

Figure 3.6: The cache layout of a binary heap.

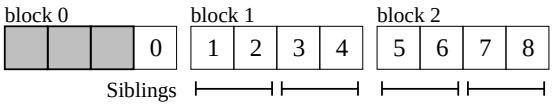

Figure 3.7: An improved cache layout of a binary heap.

unaligned layout. As the number of heap elements per cache block grows larger, the percentage of siblings that cross block boundaries is reduced and there is less inefficiency that can be eliminated. As this optimization represents a change in data layout only, it will not change the instruction cost of the heap operations.

The second observation about *remove-min* is that we want to fully utilize the cache blocks that are brought in from memory. Consider a *remove-min* operation on a cache-aligned heap that is at element 3 and is trying to decide between element 7 and element 8 as its minimum child. If four elements fit per cache block and a cache miss occurs while looking at element 7, we will have to bring the block containing elements 5-8 into the cache. Unfortunately, we will look only at two of the four elements in this cache block to find the minimum child. We are using only two elements even though we paid to bring in four.

As before, there is a straightforward solution to the problem, and the inefficiency can be eliminated by increasing the fanout of the heap so that a set of siblings fills a cache block. The *remove-min* operation will no longer load elements into the cache and not look at them. A d-heap is the generalization of a heap with fanout d rather than two. The d-heap was first introduced by Johnson as a way to reduce the cost of the *Reduce-Min* operation [Johnson 75]. Naor *et al.* suggest using d-heaps with large d as a way to reduce the number of page faults that heaps incur in a virtual memory environment [Naor et al. 91]. Figure 3.8 shows a 4-heap and the way it lays in the cache when four elements fit per cache block.

Unlike the previous optimization, this change will definitely have an impact on the instruction cost of heap operations. The *add* operation should strictly benefit from an increased fanout. *Add*s percolate an element from the bottom up and look at only one element per heap level. Therefore the shallower tree that results from a larger fanout will cause the *add* operation to execute fewer instructions. The instruction

cost of the *remove-min* operation, on the other hand, can be increased by this change. In the limit as d grows large, the heap turns into an unsorted array which requires a linear number of comparisons. Recall that the *remove-min* operation moves the last element of the array to the root, and then for each level finds the minimum of the d children and swaps this smallest element with its parent. Since the children are stored in an unsorted manner, d comparisons must be performed to find the minimum of the d children and the parent. The cost of a swap can vary depending on how much data is stored with each element. I give the swap a cost of a relative to the cost of a comparison. Thus, the total cost at each level is $d + a$. I estimate the total cost of *remove-min* as $d + a$ multiplied by the number of levels traversed. In this simple analysis, I assume that the tail element is always percolated back down to the lowest level in the heap. The total expected cost for *remove-min* counting swaps and comparisons is

$$(d + a) \log_d((d - 1)N + 1) \approx$$

$$(d + a) \log_d(dN) =$$

$$(d + a) \log_d N + (d + a) =$$

$$\frac{(d+a)}{log_2 d} \log_2 N + d + a$$

For large N, the *remove-min* cost is proportional to $log_2 N$ by a factor of $(d + a)/\log_2 d$. This factor is made up of two components: the cost of the comparisons and the cost of the swaps. Increasing d increases the time spent comparing children $(d/log_2 d)$. Increasing d also reduces the total cost of the swaps $(a/log_2 d)$. Figure 3.9 shows a graph of $(d + a)/\log_2 d$ for various values of a. For *remove-min* operations with a swap cost of at least one comparison, increasing fanout initially reduces swap

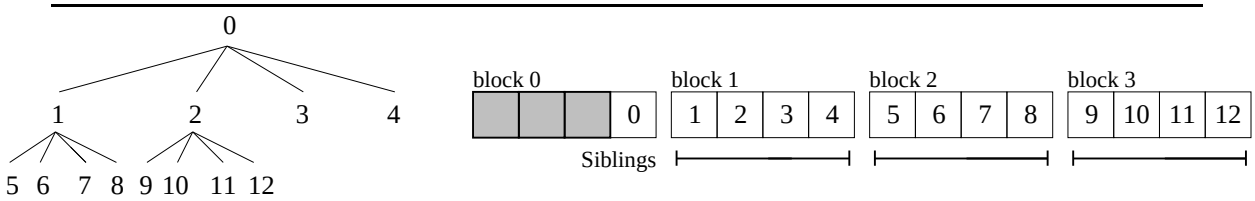

Figure 3.8: The layout of a 4-heap padded for cache alignment.

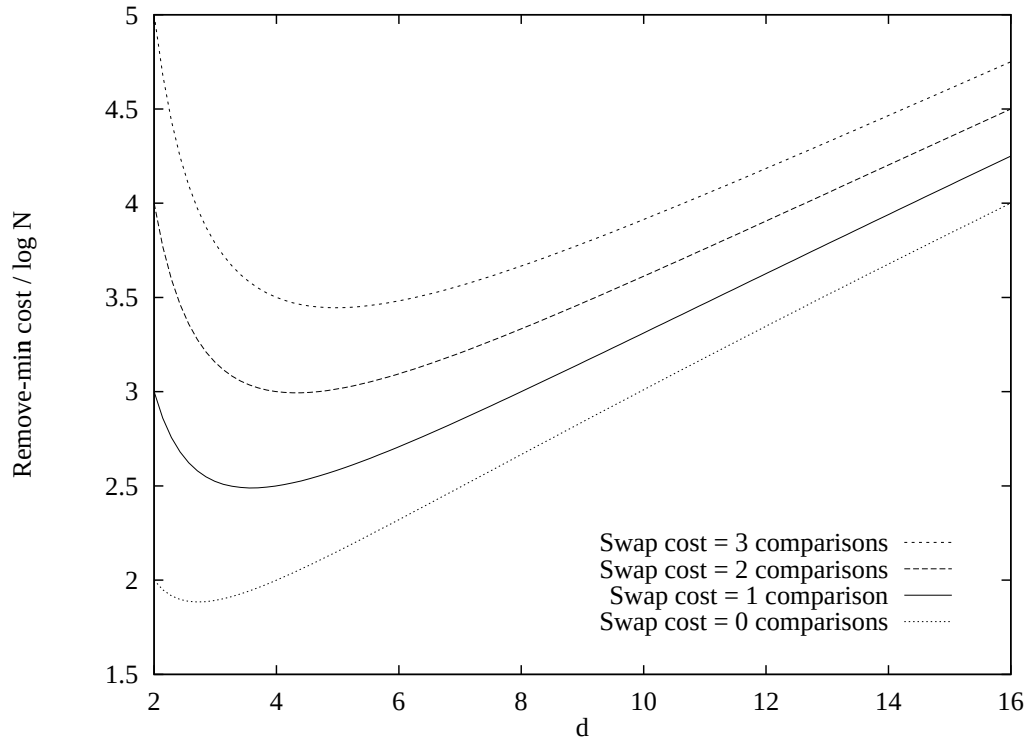

Figure 3.9: *Remove-min* cost as a function of d.

costs by more than it increases comparison costs, resulting in an overall speedup. For a swap cost of two comparisons, the *remove-min* cost is reduced by a quarter by changing the fanout from two to four and does not grow to its initial level until the fanout is larger than twelve. As long as fanouts are kept small, the instruction cost of *remove-min* should be the same or lower than for heaps with fanout two.

This graph also points out the dangers of an analysis that only considers one type of operation. It clearly shows that even if caching is not considered, a heap with fanout four should perform better than a heap with fanout two. This would not be evident however, if the number of comparisons performed were the only factor considered, as is commonly done. The curve on the graph which has swap cost of zero is the equivalent of counting only comparisons. This curve does not take swap costs into account and suggests the misleading conclusion that larger fanouts are not beneficial. Due to the prevalence of this style of analysis, the only reference I found that proposed

larger fanout heaps due to their reduced instruction cost was an exercise by Knuth in which he shows that 3-heaps perform better than 2-heaps [Knuth 73, Ex. 28 Pg. 158]. Knuth's analysis is performed in the MIX model [Knuth 73], and all instruction are accounted for, thus the benefits of larger fanouts are properly predicted.

3.4 Evaluation

To understand the impact these optimizations have on both memory system and overall performance, I measured the dynamic instruction count, the cache misses incurred and the execution time of both traditional heaps and cache-aligned d-heaps. Dynamic instruction counts and cache performance data were collected using Atom [Srivastava & Eustace 94]. Executions were run on a DEC Alphastation 250. The simulated cache was direct-mapped with a total capacity of two megabytes with a 32 byte block size, the same as the second level cache of the Alphastation 250.

3.4.1 Heaps in the Hold Model

Heaps are often used in discrete event simulations as a priority queue to store the events. In order to measure the performance of heaps operating as an event queue, I first test the heaps using the hold model [Jones 86]. In the hold model, the heap is initially seeded with some number of keys. The system then loops repeatedly, each time removing the minimum key from the heap, optionally performing some other outside work, and finally adding a random value to the element's key and adding the element back into to the heap. It is called the hold model because the size of the heap holds constant over the course of the run.

The heaps were initially seeded with uniformly distributed keys, and the heaps were run in the hold model for 3,000,000 iterations to allow the system to reach steady state. The performance of the heaps were then measured for 200,000 iterations. I first examine the performance of heaps in the hold model when no outside work is performed between iterations. I then measure the cache interactions that occur when outside work occurs between iterations.

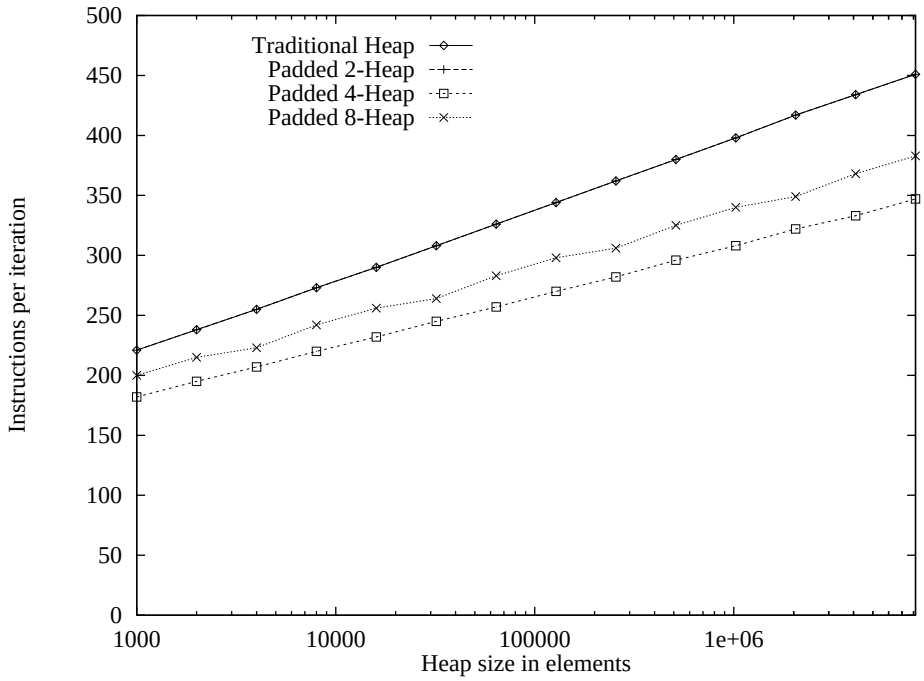

Figure 3.10: Instruction counts for heaps in the hold model with no outside work.

No Outside Work

Figure 3.10 shows the number of instructions executed per iteration for a traditional heap and an aligned 2, 4 and 8-heap running in the hold model with no outside work. The heap size is varied from 1,000 to 8,192,000. The number of instructions per element for all four heaps grows with the log of the number of elements as expected. The traditional heap and the aligned 2-heap execute the same number of instructions since their only difference is in the layout of data. Changing the fanout of the heap from two to four provides a sizable reduction in instruction cost as Figure 3.9 predicts. Also as predicted, changing the fanout from four to eight increased the instruction cost, but not higher than the heaps with fanout two.

The wobble in the curve for the 8-heap occurs because a heap's performance is based in part on how full the bottom heap level is. This graph plots the performance of heaps that successively double in size. In the case of the 2-heap, doubling the size always adds a new heap level. With the 8-heap on the other hand doubling the

number of elements may only change how full the bottom level is. The result is that the curve for the 8-heap has a wobble with a period of three data points. There is a similar wobble of period two for the 4-heap, but it is far less noticeable.

Figure 3.11 shows the number of cache misses per iteration for a 2 megabyte direct-mapped cache with a 32 byte block size and 4 byte heap elements (8 heap elements fit in a cache block). With this cache size and heap element size, 524,288 heap elements fit in the cache. Since no outside work occurs, heaps up to this size incur no cache misses once the system has warmed up. When the heap is larger than 524,288 elements, cache misses occur and the impact of the memory optimizations can be seen. This graph shows that cache aligning the traditional heap reduces the cache misses by around 15%. Increasing the heap fanout from two to four provides a large reduction in cache misses, and increasing from four to eight reduces the misses further still. This graph serves as a good indicator that the optimizations provide a significant improvement in the memory system performance of heaps.

Figure 3.12 shows the execution time on an Alphastation 250 for heaps in the hold model. No outside work is performed and the heap element size is 4 bytes. The cache-aligned 4-heap significantly outperforms both binary heaps. The low instruction cost results in a lower execution time initially and the difference in cache misses causes this gap to grow as heap size is increased. The 8-heap initially starts out slower than the 4-heap due to increased instruction cost. This difference is eventually overcome by the difference in cache misses and the 8-heap performs best for large heap sizes. The cache-aligned 2-heap did not perform significantly better than the traditional heap in this experiment. For 8,192,000 elements, the 4-heap and the 8-heap exhibit speedups of up to 46% and 57% respectively over the traditional heap.

Outside Work

In the previous experiments, heaps were run in the hold model with no outside work. In these runs none of the heap's data was ejected from the cache unless the heap itself ejected it. In reality, concurrently executing algorithms interact in the cache. Even though two algorithms might exhibit good cache behavior running alone, composing them may result in a single algorithm with bad cache behavior. Accordingly, I now measure the performance of heaps in the hold model when outside work is performed between the *remove-min* and the *add* in each iteration. The outside work performed

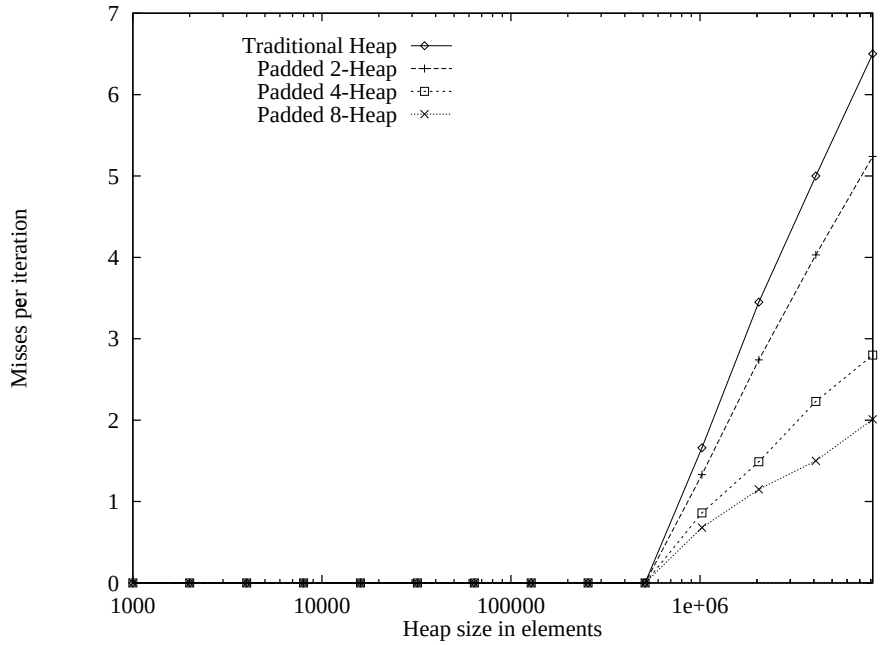

Figure 3.11: Cache performance of heaps in the hold model with no outside work. Simulated cache size is 2 megabytes, block size is 32 bytes and 8 elements fit per block.

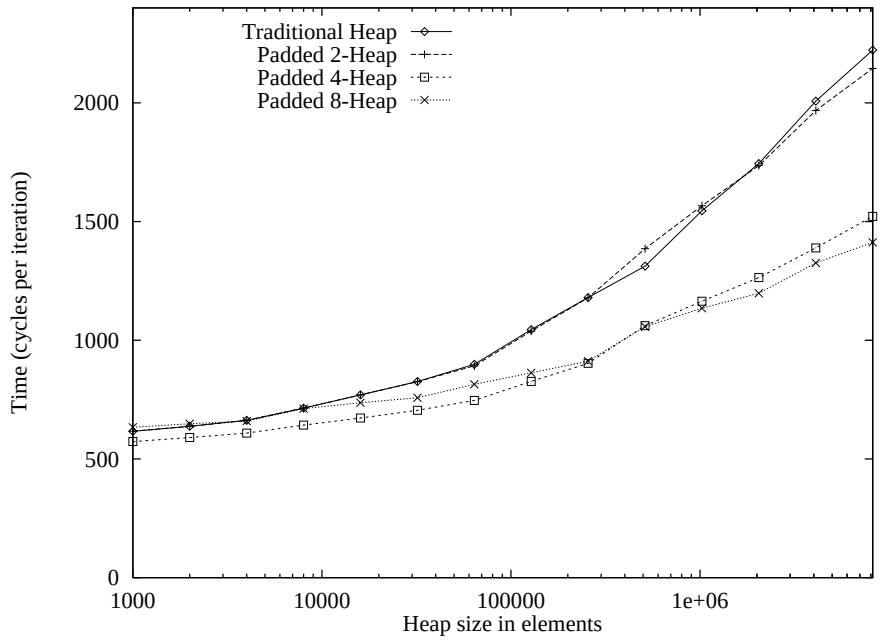

Figure 3.12: Execution time for heaps in the hold model with no outside work. Executions on a DEC Alphastation 250 with 8 elements per block.

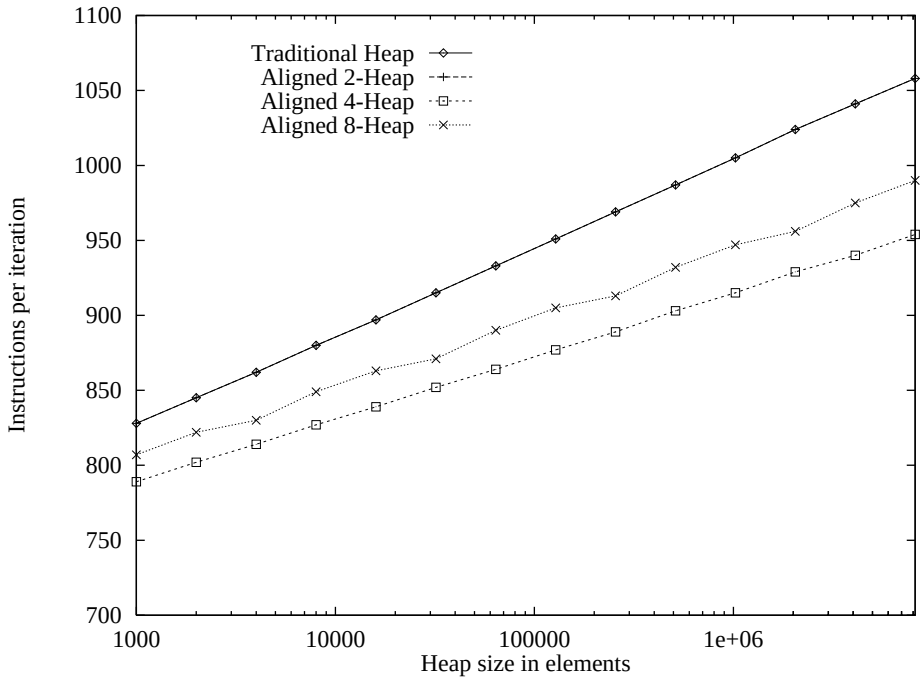

Figure 3.13: Instruction counts for heaps in the hold model with outside work.

in these experiments consists of 25 uniformly-distributed random reads from a two megabyte array. While this probably does not model a typical simulation event handler, it is a reasonable approximation of outside work and helps to show the cache interference between independent algorithms.

Figure 3.13 shows the dynamic instruction count for a traditional heap, and an aligned 2, 4 and 8-heap running in the hold model with the outside work consisting of 25 random reads. The heap size is varied from 1,000 to 8,192,000. This graph is the same as the previous instruction count graph with the exception that it is shifted up due to the cost of performing the outside work. As before, the binary heaps execute the same number of instructions, with the 8-heap performing fewer and the 4-heap least of all.

Figure 3.14 shows the number of cache misses incurred per iteration for the heaps when eight heap elements fit in a cache block. This graph shows that in contrast to the previous experiments, the heaps incur cache misses even when the heap is smaller than the cache. This is due to the interaction between the outside work and the heap,

and this interaction is significant. When outside work is present, the traditional heap of size 8,192,000 incurs 17.1 cache misses per iteration versus only 6.4 when there is no outside work. Again, we see that the optimizations significantly reduce cache misses. For large heaps, the aligned 2, 4 and 8-heaps incur 15%, 49% and 62% fewer cache misses than the traditional heap respectively.

Figure 3.15 shows the execution time for heaps in the hold model with outside work. This graph looks very much like a simple combination of the instruction count and cache miss graphs. Aligning the traditional heap provides a small reduction in execution time. Increasing the fanout from two to four provides a large reduction in execution time due to both lower instruction cost and fewer cache misses. The 8-heap executes more instructions than the 4-heap and consequently executes slower initially. Eventually the reduction in cache misses overcomes the difference in instruction cost, and the 8-heap performs best for large data sets. For 8,192,000 elements, the 4-heap and the 8-heap exhibit speedups of up to 22% and 28% respectively over the traditional heap.

To this point, I have only considered heap configurations for which a set of siblings fit in a single cache block. The result is that if the heap is cache-aligned, only one cache block needs to be read per heap level. If the size of a set of siblings grows larger than the cache, we will need to load at least two cache blocks per heap level. I expect that this will result in a increase in the number of cache misses per operation and a decrease in overall performance. To test this, I measure the performance of a traditional heap and a 2, 4 and 8-heap when only 4 heap elements fit per cache block. Heap size was varied from 1,000 to 4,096,000 elements and outside work consisted of 25 random reads per iteration.

The only change between this experiment and the previous is the size of the heap elements. As a result, the number of instructions executed per iteration is unchanged from Figure 3.13. The main performance change is in the number of cache misses incurred. Figure 3.16 shows the cache performance of heaps when 4 heap elements fit in a cache block. The first thing to note is the larger gap between the traditional heap and the aligned 2-heap. Since fewer heap elements fit per cache block, a higher percentage of siblings cross cache block boundaries in the traditional heap. The result is that aligning the traditional heap eliminates more misses than it did when 8 elements fit per block. As before, the 4-heap incurs fewer misses than the binary

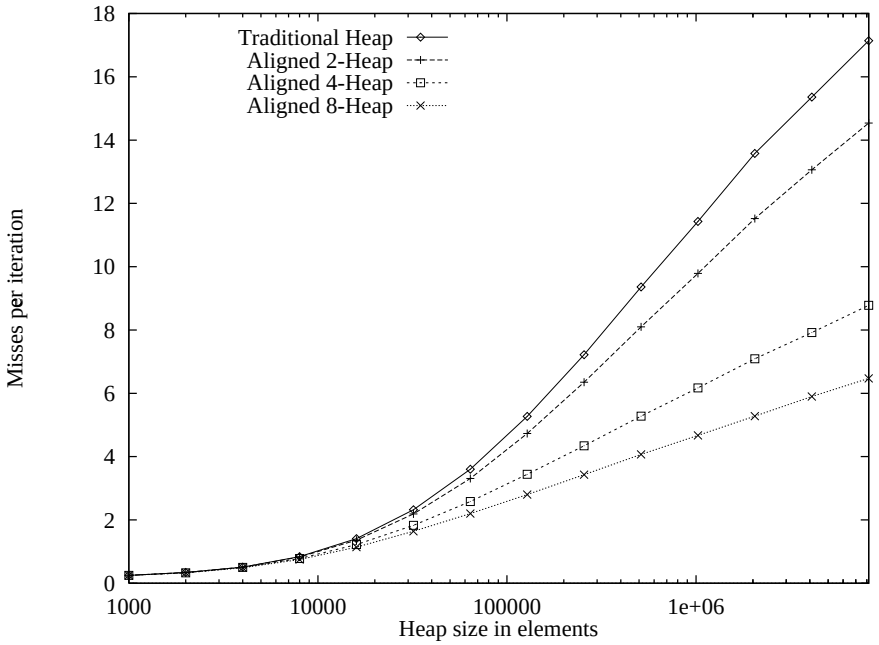

Figure 3.14: Cache performance of heaps in the hold model with outside work. Simulated cache size is 2 megabytes, block size is 32 bytes and 8 elements fit per block.

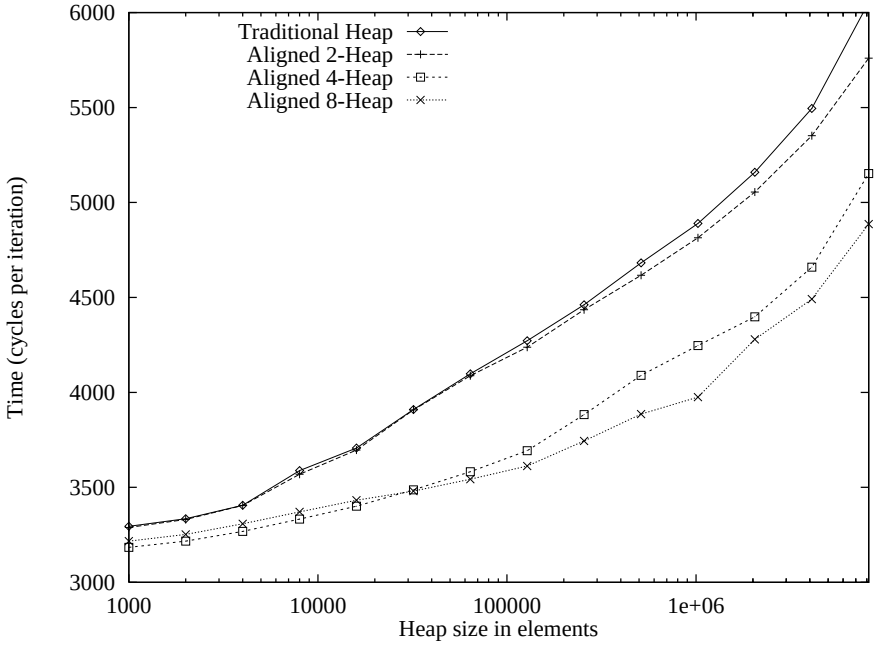

Figure 3.15: Execution time for heaps in the hold model with outside work. Executions on a DEC Alphastation 250 with 8 elements per block.

heaps. The graph shows that the 8-heap incurs more misses than the 4-heap. While the 8-heap has a shallower tree to traverse, it incurs more misses per heap level, resulting in worse cache performance than the 4-heap.

Figure 3.17 shows the execution times for heaps when 4 elements fit per cache block. In this graph, the 4-heap outperforms all other heaps for all data set sizes. The only difference between this graph and Figure 3.15 is the number of heap elements that fit per cache block. These graphs show how architectural constants can be used to configure an algorithm for the best performance.

The Interaction of Algorithms in the Cache

When two algorithms are composed together, the instruction count of the resulting algorithm is simply the instruction count of the two algorithms added together. The cache performance of the resulting algorithm, however, is not as easy to predict. Since the performance of a memory reference is dependent on the previous references and combining algorithms alters the sequence of memory references, algorithms can combine in complicated ways, and the resulting algorithm's cache performance is not obvious.

An interesting aspect of this interaction is that memory optimizations can potentially have a more significant impact on performance than traditional instruction cost optimizations. For instance, if a particular algorithm accounts for only 10% of the instructions executed, optimizing this algorithm can eliminate at most 10% of the total instruction count. This is not the case with cache misses, and optimizing an algorithm can potentially eliminate more than its share of the total cache misses incurred by the system. This effect is demonstrated in the cache simulations of the heaps operating in the hold model with outside work. When there are 8,192,000 elements in the heap and 8-heap elements fit per cache block, the traditional heap incurs a total of 17.1 misses per iteration while the 8-heap incurs 6.4 (Figure 3.14). A division of these misses between the heap and the outside work is shown in Table 3.1. The first row shows that with a traditional heap, 9.6 misses per iteration occur while accessing the heap and 7.5 occur while accessing the work array. Replacing the traditional heap with an 8-heap reduces the misses to 3.3 per iteration, but it also reduces the misses of the unaltered outside work process to 3.1 per iteration. Even though the heap only accounts for 56% of the total misses, optimization the heap reduces

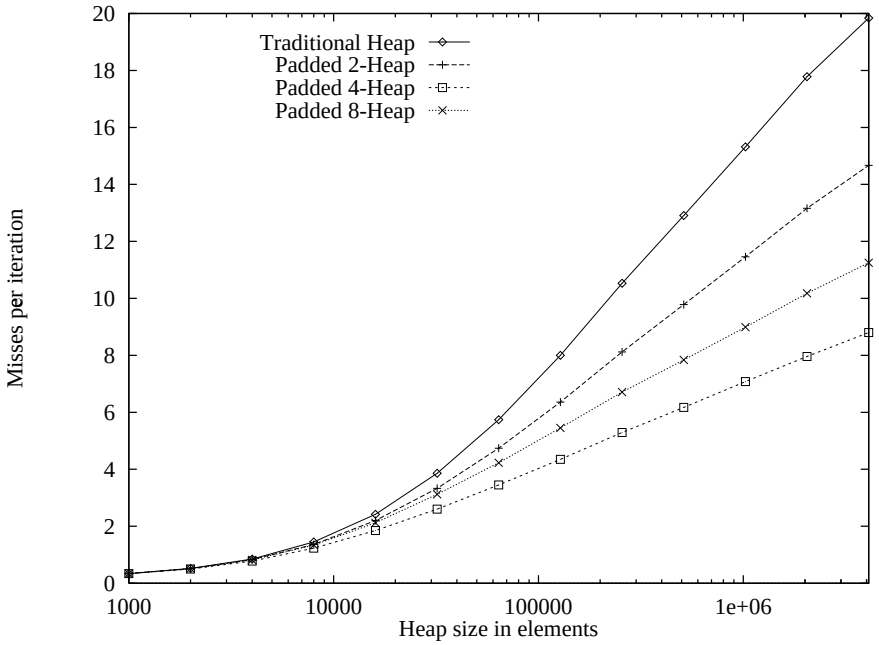

Figure 3.16: Cache performance of heaps in the hold model with outside work. Simulated cache size is 2 megabytes, block size is 32 bytes and 4 elements fit per block.

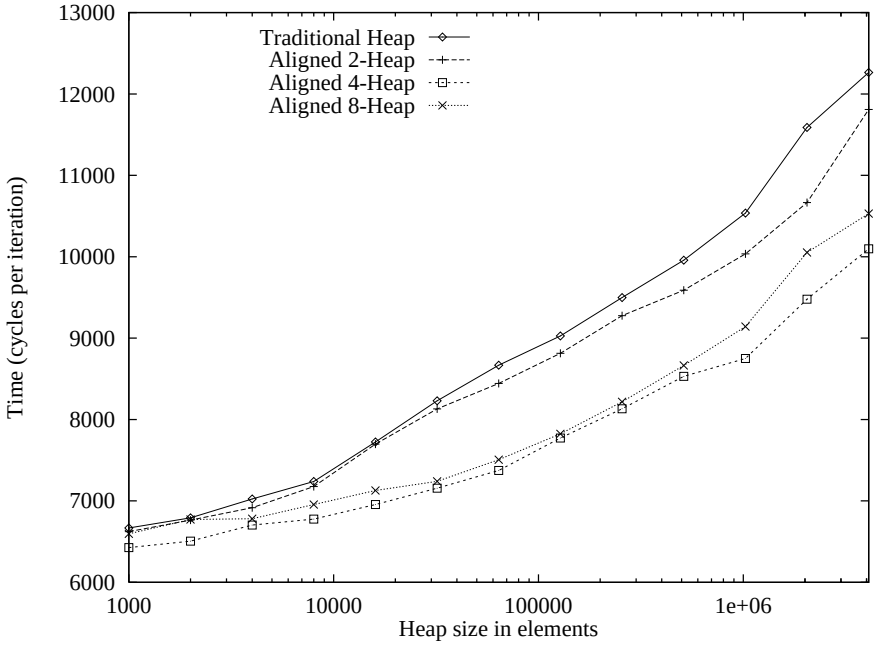

Figure 3.17: Execution time for heaps in the hold model with outside work. Executions on a DEC Alphastation 250 with 4 elements per block.

Table 3.1: Division of cache misses between heap and outside work process with 8,192,000 elements.

Method	Misses for Heap		Misses for Outside Work		Misses Total
Traditional Heap	9.6	56%	7.5	44%	17.1
8-heap	3.3	52%	3.1	48%	6.4

misses from 17.1 to 6.4, a drop of 63%. This demonstrates that optimizing an algorithm can improve both its own cache performance and the cache performance of the algorithms it interacts with. This is an important effect and should be taken into account when considering the potential impact of a memory optimization on overall cache performance.

3.4.2 Heapsort

Heaps are the core of the heapsort algorithm [Williams 64]. In Chapter 6, I closely examine the memory system performance of a number of sorting algorithms, including heapsort. For now, I provide only a graph of execution time to demonstrate the effects the optimizations from this chapter have on the overall performance of heapsort. Figure 3.18 shows the execution time of heapsort built from four different heaps running on a DEC Alphastation 250 sorting uniformly distributed 32 bit integers. As before, we see that increasing fanout from two to four provides a large performance gain. Again we see that the instruction count overhead of the 8-heap is overcome by the reduced cache misses, and the 8-heap performs best for larger heap sizes. The 8-heap sorting 8,192,000 numbers nearly doubles the performance of the traditional binary heap.

3.4.3 Generality of Results

So far all experiments have been performed on a DEC Alphastation 250. In order to demonstrate the general applicability of these optimizations across modern architectures, I now present data for the heaps running in the hold model on four additional machines. Figure 3.19 shows the speedup of an aligned 4-heap over a traditional heap running in the hold model with 8 byte heap elements and no outside

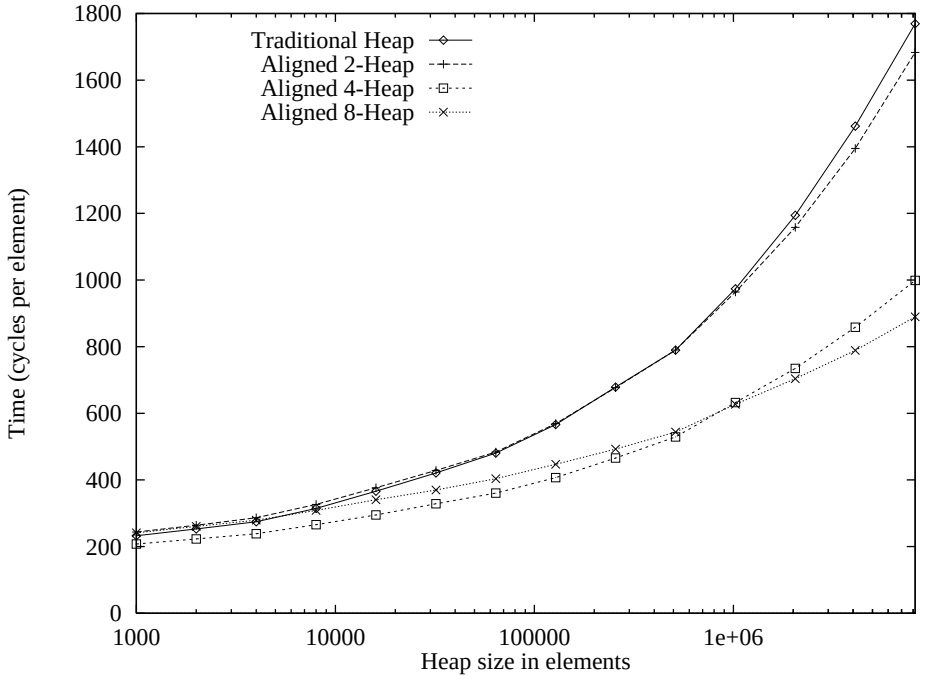

Figure 3.18: Execution time for heapsort on a DEC Alphastation 250 using 32 bit uniformly distributed integers.

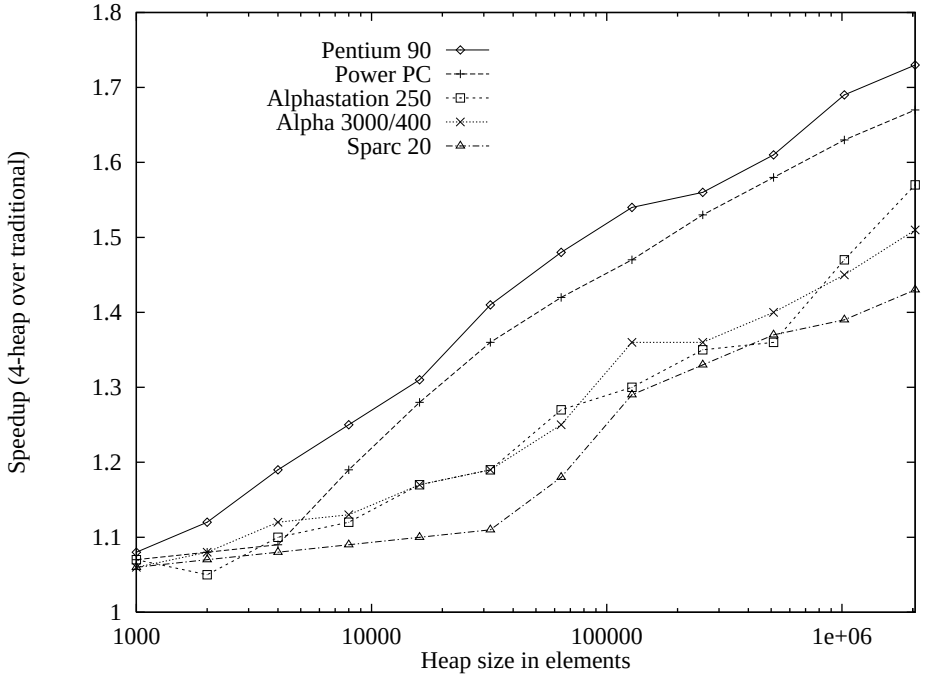

Figure 3.19: Speedup of an aligned 4-heap over a traditional heap. Executions on a variety of architectures with 4 elements per block.

Table 3.2: Clock rate and cache sizes of various machines.

Machine	Processor	Clock Rate	L1 Cache Size	L2 Cache Size
Alphastation 250	Alpha 21064A	266 MHz	8k	2048k
Pentium 90	Pentium	90 MHz	8k	256k
Alpha 3000/400	Alpha 21064	133 MHz	8k	512k
Power PC	MPC601	80 MHz	32k	512k
Sparc 20	SuperSparc	60 MHz	16k	1024k

work. The four additional machines were the Sparc 20, the IBM Power PC, the DEC Alpha 3000/400 and a Pentium 90 PC. The clock rates of the machines tested ranged from 60 to 266 megahertz, and the second level cache sizes ranged from 256k to 2048k. Table 3.2 shows the processor, clock rate and cache size for the machines I tested. Figure 3.19 shows that despite the differences in architecture, the 4-heaps consistently achieve good speedups that increase with the size of the heap.

For heaps that fit entirely in the cache, we expect that no cache misses will occur and all of the speedup will be due to the reduction in instruction cost. Larger than that size, we expect the speedup curve to climb as the memory optimizations further improve performance. For the Sparc and the Alphas, this knee in the curve occurs around the size of the second level cache as expected. For the Power PC and the Pentium, however, the speedup curve climbs almost immediately. These differences might be due in part to differences in the number of first level cache misses. This would not explain all of the difference, however, as the first level miss penalties in these machines are small.

I believe these differences are primarily due to variations in the page mapping policies of the operating systems. The early increase in speedup for the Power PC and the Pentium indicates that for these two machines, cache misses occur even when the heaps are considerably smaller than the cache. Some operating systems, such as Digital Unix, have virtual to physical page mapping polices that attempt to minimize cache misses [Taylor et al. 90]. These policies try to map pages so that blocks of memory nearby in the virtual address space do not conflict in the cache. The operating systems for the Power PC and the Pentium (AIX and Linux respectively)

appear not to be as careful. The result is that even for small heaps, the allocated memory conflicts in the cache, and this causes second level cache misses to occur earlier than we expect. These cache misses provide an earlier opportunity for the 4-heap to reduce misses and account for the increased speedup.

3.5 Summary

The experiments in this chapter demonstrate the potential gains of cache conscious algorithm design. Two memory optimizations of implicit heaps are motivated and developed. The first optimization adjusts the data layout of heaps in order to prevent heap siblings from crossing cache block boundaries. The second optimization increases the fanout of the heap in order to improve spatial locality. The effectiveness of these optimizations is tested by running a heap in the hold model and by running heapsort. Cache simulations are used to demonstrate that these optimizations significantly reduce the cache misses that heaps incur, and executions on a Alphastation 250 show that these reductions in cache misses translate into in a corresponding improvement in overall performance. Finally, to demonstrate the general applicability of these optimizations, heap performance was compared for five architectures with varying memory systems. The memory-tuned heaps consistently outperformed the traditional heaps in the hold model, and for all five architecture the memory-tuned heaps provided speedups that increased as the heap size was increased.

Chapter 4

Collective Analysis

While simple and accurate, trace driven simulation does not offer the benefits of an analytical cache model, namely the ability to quickly obtain estimates of cache performance for varying cache and algorithm configurations. An analytical cache model also has the inherent advantage that it helps a designer understand the algorithm and helps suggest possible optimizations. As we saw in Chapter 3, the cache interactions between concurrently executing algorithms can be substantial. For this reason, it is important that an analytical model allow a system of algorithms to be analyzed collectively as well as individually. Towards this goal, I develop *collective analysis*, a framework for predicting the cache performance of a system of algorithms. Collective analysis allows algorithms to be analyzed without implementations and without address traces. In this way, collective analysis is more like the purely analytical models of Temam *et al.* than the hybrid trace-driven techniques of Agarwal *et al.* and Singh *et al.* [Temam et al. 94, Agarwal et al. 89, Singh et al. 92].

The main difference between the work in this chapter and Temam's work is the assumption made regarding the algorithm's memory reference pattern. Temam's model assumes that the exact reference pattern of the algorithm can be determined at analysis time, and no data dependent array accesses or control flow are allowed. Collective analysis, on the other hand, is intended for algorithms whose general memory behavior is known but whose exact reference pattern is not. The result is that the two techniques apply to different classes of algorithms. For algorithms with non-oblivious reference patterns, collective analysis can be applied and Temam's model cannot. The performance of algorithms with oblivious reference patterns, however, will be more accurately predicted using Temam's model.

In order to demonstrate its use, I apply collective analysis to cache-aligned and unaligned d-heaps operating in the hold model. I validate the accuracy of the analysis by comparing the predictions of collective analysis with the trace driven simulation results from Chapter 3.

4.1 The Memory Model

Collective analysis is a framework that can be used to predict the cache performance of a system of algorithms in a realistic memory model. Collective analysis assumes that there is a single cache with a total capacity of C bytes, where C is a power of two. The cache has a block size of B bytes, where $B \leq C$ and is also a power of two. In order to simplify analysis, collective analysis only models direct mapped caches [Hennesey & Patterson 90] and does not distinguish reads from writes. It assumes that items that are contiguous in the virtual address space map to contiguous cache locations, which means that it models a virtually indexed cache [Hennesey & Patterson 90]. The memory model does not include a TLB, nor does it attempt to capture page faults due to physical memory limitations.

4.2 Applying the Model

The goal of collective analysis is to approximate the memory behavior of an algorithm and predict its cache performance characteristics from this approximation. The first step is to partition the cache into a set of *regions* R, where regions are non-overlapping, but not necessarily contiguous, sections in the cache. All cache blocks must belong to a region, and a cache block cannot be split across regions. The cache should be divided into regions in such a way that the accesses to a particular region are uniformly randomly distributed across that region. If the accesses are not uniformly distributed, the region should be subdivided into multiple regions that do have uniform access patterns. Once the accesses within a region are uniformly distributed, further subdivision should be avoided in order to minimize the complexity of the analysis.

The next step is to break the system of algorithms to be analyzed into a set of independent stochastic *processes* P, where a process is intended to characterize the memory behavior of an algorithm or part of an algorithm. The behavior of the system

need not be represented exactly by the processes, but any simplifying approximations are made at the expense of corresponding inaccuracies in the results. Areas of the virtual address space accessed in a uniform way should be represented with a single process. Collectively, all of the processes represent the accesses to the entire virtual address space and hence represent the system's overall memory behavior.

Collective analysis assumes that the references to memory satisfy the *independent reference assumption* [Coffman & Denning 73]. In this model each access is independent of all previous accesses; that is, the system is memoryless. Algorithms which exhibit very regular access patterns such as sequential traversals will not be accurately modeled in this framework because of the independent reference assumption.

I define the *access intensity* of a process or set of processes to be the rate of memory accesses per instruction by the process or set of processes. Let λ_{ij} be the access intensity of process j in cache region i. Let λ_i be the access intensity of the set of all processes in cache region i. Let λ be the access intensity of all the processes in all the regions. Given the assumption about uniform access, if a cache block b is in region i and the region contains m cache blocks, then process j accesses block b with intensity λ_{ij}/m. Once the system has been decomposed into processes and the intensities have been expressed, the calculations to predict cache performance are fairly simple.

The total access intensity by the system for cache region i is

$$\lambda_i = \sum_{j \in P} \lambda_{ij}.$$

The total access intensity of the system is:

$$\lambda = \sum_{i \in R} \lambda_i = \sum_{i \in R} \sum_{j \in P} \lambda_{ij}. \tag{4.1}$$

In an execution of the system a *hit* is an access to a block by a process where the previous access to the block was by the same process. An access is a *miss* if it is not a hit. The following theorem is a reformulation of the results of Rao [Rao 78]. This formulation differs from Rao's only in that blocks are grouped together into a region if the accesses are uniformly distributed in the region. This simplifies the analysis considerably.

Theorem 1 *The expected total hit intensity of the system is*

$$\eta = \sum_{i \in R} \frac{1}{\lambda_i} \sum_{j \in P} \lambda_{ij}^2. \tag{4.2}$$

Proof: Define η to be the expected total hit intensity of the system and η_i to be the expected hit intensity for region i. The total hit intensity of the system is the sum of the hit intensities for each region. The hit intensity for a region is the sum across all processes of the hit intensity of the process in that region.

An access to a block in a direct mapped cache by process j will be a hit if no other process has accessed the block since the last access by process j. I say that a cache block is *owned* by process j at access t if process j performed the last access before access t on the block.

Consider a block b in a region i containing m blocks. The access intensity of process j in block b is $\frac{\lambda_{ij}}{m}$ and the total intensity in block B is $\frac{\lambda_i}{m}$. Hence on average, block b is owned by process j $\frac{\lambda_{ij}}{\lambda_i}$ of the time. The hit intensity of process j in block b is $\frac{\lambda_{ij}}{\lambda_i}\frac{\lambda_{ij}}{m}$, and the expected hit intensity for the entire region i by process j is $\frac{\lambda_{ij}^2}{\lambda_i}$. Hence, $\eta_i = \frac{1}{\lambda_i} \sum_{j \in P} \lambda_{ij}^2$ and $\eta = \sum_{i \in R} \frac{1}{\lambda_i} \sum_{j \in P} \lambda_{ij}^2$.

□

Corollary 1 *The expected overall miss intensity is* $\lambda - \eta$.

Corollary 2 *The expected overall hit ratio is* $\frac{\eta}{\lambda}$.

4.3 A Simple Example

In order to illustrate the use of collective analysis, I introduce a simple algorithm which uses two data-structures: one array the size of the cache, and another array one-third the size of the cache. The algorithm loops forever, and each time through the loop it reads two values at random from the large array, adds them together and stores the result in a randomly selected element of the small array.

I begin by decomposing the cache into two regions. Region 0 is two-thirds the size of the cache and only holds blocks from the large array. Region 1 is one-third the size of the cache, and holds blocks from both the large and the small array.

I divide the system into two processes: process 0 for the reads from the large array and process 1 for the writes to the small array. First consider the accesses performed by process 1. Since process 1 never makes accesses outside of the small array, I know that λ_{01} is 0. Let μ be the loop's rate of execution. Process 1 accesses the small array once every time around the loop, so λ_{11} is μ. Since there are twice as many reads from the big array as writes to the small array, the total access intensity for process 0 is 2μ. Since the reads are evenly distributed across the big array, we expect that two-thirds of the reads go to region 0 and one-third go to region 1, thus λ_{00} is $\frac{4}{3}\mu$ and λ_{10} is $\frac{2}{3}\mu$. The overall hit ratio is easily computed:

$$\lambda_{00} = \tfrac{4}{3}\mu, \lambda_{01} = 0, \lambda_{10} = \tfrac{2}{3}\mu, \lambda_{11} = \mu$$

$$\lambda_0 = \tfrac{4}{3}\mu + 0 = \tfrac{4}{3}\mu$$

$$\lambda_1 = \tfrac{2}{3}\mu + \mu = \tfrac{5}{3}\mu$$

$$\lambda = \sum_{i \in R} \lambda_i = \frac{4}{3}\mu + \frac{5}{3}\mu = 3\mu$$

$$\eta = \sum_{i \in R} \frac{1}{\lambda_i} \sum_{j \in P} \lambda_{ij}^2 = \frac{(\frac{4}{3}\mu)^2}{\frac{4}{3}\mu} + \frac{0^2}{\frac{4}{3}\mu} + \frac{(\frac{2}{3}\mu)^2}{\frac{5}{3}\mu} + \frac{\mu^2}{\frac{5}{3}\mu} = \frac{33}{15}\mu$$

$$\text{Overall cache hit ratio} = \frac{\eta}{\lambda} = \frac{\frac{33}{15}\mu}{3\mu} = \frac{11}{15} \cong 73\%$$

While this example is simple, the next section shows how collective analysis can be used to predict the performance of more complex systems.

4.4 Cache-Aligned d-heaps

In this section, I analyze the cache performance of d-heaps in the hold model when sets of siblings to not cross cache block boundaries. Recall that a d-heap is a generalization of a binary heap with fanout d rather than two. A d-heap with N elements has depth $\lceil \log_d((d-1)N + 1) \rceil$. The elements have the following relationships

$$Parent(i) = \lfloor \frac{i-1}{d} \rfloor$$

$$Children(i) = di + 1, di + 2, \ldots, di + d$$

As with binary heaps, d-heaps must satisfy the *heap property*, which says that for all elements except the root, $Key[Parent(i)] \leq Key[i]$. It follows that the minimum key in the data structure must be at the root. Let e be the size in bytes of each heap element.

The hold model repeatedly removes an element from the heap, performs outside work, and adds the element back to the heap. In the hold model, the cache performance of the *add* operation is easy to predict. Since the heap is always the same size when the *add* is performed, it always uses the same chain of elements from leaf to root. This chain of elements will quickly be brought into the cache and will seldom be ejected once the heap reaches steady state. Thus, the number of misses incurred by the *add* operation is small enough that it can be ignored. The behavior of the *remove-min* and the outside work is more complicated, and collective analysis can be used to understand their performance.

In this analysis, I restrict heap configurations to those in which all of a parent's children fit in a single cache block (where $de \leq B$). This limits the value of d; for a typical cache block size of 32 bytes, fanout is limited to 4 for 8 byte heap elements, and fanout is limited to 8 for 4 byte heap elements. I also restrict the analysis to heap configurations in which the bottom layer of the heap is completely full (*i.e.* where $\lceil \log_d((d-1)N+1) \rceil = \log_d((d-1)N+1)$).

In order to predict cache misses for d-heaps in the hold model with the same outside work performed in Chapter 3, I model the work performed between the *remove-min* and the *add* as w random uniformly distributed accesses to an array of size C. Leaving w unbound in the analysis allows me to compare the model predictions to the simulation results from Chapter 3 both with outside work ($w = 25$) and without outside work ($w = 0$).

The first step of the analysis is to divide the cache into regions based on the heap's structure. Recall that we have a cache with a size of C bytes, a cache block size of B bytes, and a heap with N elements, fanout d, and element size e. Let $S = \frac{C}{e}$ be the size of the cache in heap elements.

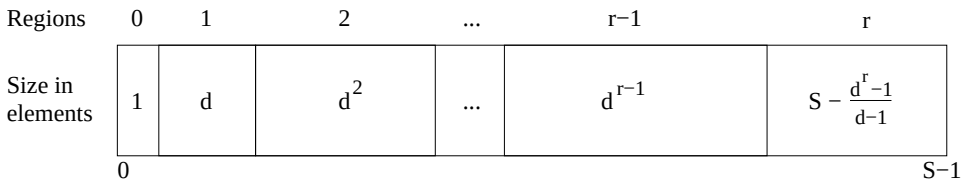

Figure 4.1: The division of the cache into regions for the d-heap.

Given the heap's structure, $r = \lfloor \log_d((d-1)S+1) \rfloor$ whole levels of the heap will fit in the cache. That is to say, heap levels $0 \ldots r-1$ fit in the cache and heap level r is the first level to spill over and wrap around in the cache. I divide the cache into $r+1$ regions, where regions $0 \ldots r-1$ are the same size as the first r heap levels and region r takes up the remaining space in the cache. I define S_i to be the size of cache region i. Region i, $0 \leq i < r$, is of size d^i and region r is of size $S - \frac{d^r-1}{d-1}$. In this analysis of d-heaps, the cache regions are all contiguous; Figure 4.1 shows the division.

The next step is to partition the system into processes which approximate its memory access behavior. It is at this point that I make three simplifying assumptions about the behavior of the *remove-min* operation. I first simplify the percolating down of the tail element by assuming that all levels of the heap are accessed independently, once per *remove-min* on average. I also assume that when a heap level is accessed by *remove-min*, all sets of siblings are equally likely to be searched through for the minimum. While this is clearly not how the *remove-min* algorithm behaves, the rates of accesses and overall access distributions should be reasonably accurate.

To further simplify the heap's behavior, I make a final assumption regarding the caching of elements. In the event that a set of siblings is brought into the cache on a miss, other adjacent sets of siblings may be read in at the same time (if $de < B$). The result is that a reference to these other sets of siblings may not incur a fault even though they have never been accessed before. In this analysis, I ignore this effect and assume that neighboring sets of siblings are never faulted in. The performance results at the end of this chapter suggest that this assumption has little effect on the accuracy of the predictions.

The basic structure of the decomposition is to create one or more processes for each level in the heap. Given a total of N elements, there are $t = log_d((d-1)N+1)$ levels in the heap. Let N_i be the size in elements of heap level i. I begin by dividing the t heap levels into two groups. The first group contains the first r levels $0 \ldots r-1$ of the heap that will fit in the cache without any overlap. The second group contains the remaining levels of the heap $r \ldots t-1$. For $0 \leq i < t$, $N_i = d^i$.

Lemma 4.1 *The size of heap level i, $r \leq i < t$, is a multiple of S, the size of the cache in heap elements.*

Proof: Since $N_i = d^i$, it is sufficient to prove that $d^i \bmod S = 0$ for $r \leq i < t$. Since d, C and e are positive powers of 2, and $e < B \leq C$, both d^i and S are also positive powers of 2. It is sufficient to show that $d^i \geq S$. Let $d = 2^x$ and $S = 2^y$.

$$d^i = d^{i-r} d^r \geq d^r = d^{\lfloor \log_d((d-1)S+1) \rfloor} \geq d^{\lfloor \log_d(\frac{dS}{2}) \rfloor} =$$

$$2^{x \lfloor log_{2^x}(2^{x+y-1}) \rfloor} = 2^{x \lfloor \frac{x+y-1}{x} \rfloor} \geq 2^{x(\frac{x+y-1}{x} - \frac{x-1}{x})} = 2^y = S.$$

□

For each heap level i in the first group, I create a process i, giving us processes $0 \ldots r-1$. For the second group I create a family of processes for each heap level, and each family will have one process for each cache-sized piece of the heap level. For heap level i, $r \leq i < t$, I create $\frac{N_i}{S}$ processes called $(i,0) \ldots (i, \frac{N_i}{S} - 1)$.

Finally, I create a process to model the w random accesses that occur between the *remove-min* and the *add*. I call this process o.

If *remove-min* is performed on the heap at an intensity of μ the access intensities are

$$\lambda_{ij} = \begin{cases} \mu & \text{if } 0 \leq j < r \text{ and } i = j, \quad (4.3) \\ \frac{S_i}{N_x}\mu & \text{if } j = (x,y) \text{ and } r \leq x < t \text{ and } 0 \leq y < \frac{N_x}{S}, \quad (4.4) \\ \frac{S_i}{S} w\mu & \text{if } j = o, \quad (4.5) \\ 0 & \text{otherwise.} \quad (4.6) \end{cases}$$

In the simplified *remove-min* operation, heap levels are accessed once per *remove-min* on average, and each access to a heap level touches one cache block. An access by process j, $0 \leq j < r$, represents an access to heap level j since we know that all accesses by process j will be in cache region j, and process j makes no accesses outside of cache region j. Thus, with a *remove-min* intensity of μ, process j, $0 \leq j < r$, will access region i with an intensity of μ if $i = j$ and 0 otherwise (Equations 4.3 and 4.6).

Next, consider a process (x, y) where $r \leq x < t$. This process represents one of $\frac{N_x}{S}$ cache-sized pieces from heap level x. We expect that one cache block will be accessed from heap level x per *remove-min*. The chance that the block accessed belongs to the process in question is $\frac{S}{N_x}$. The total access intensity of the process is the access intensity of the level multiplied by the chance that an access belongs to the process, or $\frac{S}{N_x}\mu$. Since the process models accesses to a piece of the heap exactly the size of the cache, it is easy to calculate the access intensities for each region. Since the accesses of process (x, y) are uniformly distributed across the cache, the access intensities for each region will be proportional to its size. Given that the process's access intensity is $\frac{S}{N_x}\mu$, the access intensity of process (x, y), where $r \leq x < t$ in cache region i, is $\frac{S}{N_x}\mu\frac{S_i}{S} = \frac{S_i}{N_x}\mu$ (Equation 4.4).

Given that the system iterates at a rate of μ, the total access intensity of process o is $w\mu$. Since process o accesses an array of size C uniformly, we expect the accesses to spread over each cache region proportionally to its size. Thus, the access intensity of process o in cache region i is $\frac{S_i}{S}w\mu$ (Equation 4.5).

The region intensities λ_i and the overall intensity are summarized below and are followed by their derivations.

$$\lambda_i = \begin{cases} \mu + \frac{S_i}{S}(t - r + w)\mu & \text{if } 0 \leq i < r, \\ \\ \frac{S_r}{S}(t - r + w)\mu & \text{if } i = r. \end{cases}$$

$$\lambda = (t + w)\mu.$$

The expected access intensity for cache region i, where $0 \leq i < r$ is

$$\lambda_i = \sum_{j \in P} \lambda_{ij} = \mu + \sum_{x=r}^{t-1} \sum_{y=0}^{\frac{N_x}{S}-1} \frac{S_i}{N_x}\mu + \frac{S_i}{S} w\mu = \mu + \frac{S_i}{S}(t - r + w)\mu$$

The expected access intensity for cache region r is

$$\lambda_r = \sum_{x=r}^{t-1} \sum_{y=0}^{\frac{N_x}{S}-1} \frac{S_r}{N_x}\mu + \frac{S_r}{S} w\mu = \frac{S_r}{S}(t - r + w)\mu$$

The total access intensity is

$$\begin{aligned}
\lambda &= \sum_{i \in R} \lambda_i = \sum_{i=0}^{r-1} (\mu + \frac{S_i}{S}(t - r + w)\mu) + \frac{S_r}{S}(t - r + w)\mu \\
&= \mu(r + \frac{1}{S}(t - r + w)\frac{d^r - 1}{d - 1} + \frac{1}{S}(t - r + w)(S - \frac{d^r - 1}{d - 1})) \\
&= (t + w)\mu
\end{aligned}$$

An expression for the hit intensity η can be derived from Equation 4.2. The sum across regions is broken into $0 \ldots r-1$ and r, and the sum across processes is broken up based on the two groups of heap levels.

$$
\begin{aligned}
\eta &= \sum_{i \in R} \frac{1}{\lambda_i} \sum_{j \in P} \lambda_{ij}^2 \\
&= \sum_{i=0}^{r-1} \frac{1}{\lambda_i} \left(\sum_{j=0}^{r-1} \lambda_{ij}^2 + \sum_{x=r}^{t-1} \sum_{y=0}^{\frac{N_x}{S}-1} \lambda_{i(x,y)}^2 + \lambda_{io}^2 \right) + \frac{1}{\lambda_r} \left(\sum_{j=0}^{r-1} \lambda_{rj}^2 + \sum_{x=r}^{t-1} \sum_{y=0}^{\frac{N_x}{S}-1} \lambda_{r(x,y)}^2 + \lambda_{ro}^2 \right) \\
&= \sum_{i=0}^{r-1} \frac{1}{\lambda_i} \left(\mu^2 + \sum_{x=r}^{t-1} \sum_{y=0}^{\frac{N_x}{S}-1} (\frac{S_i}{N_x}\mu)^2 + (\frac{S_i}{S} w\mu)^2 \right) + \frac{1}{\lambda_r} \left(0 + \sum_{x=r}^{t-1} \sum_{y=0}^{\frac{N_x}{S}-1} (\frac{S^r}{N_x}\mu)^2 + (\frac{S_r}{S} w\mu)^2 \right) \\
&= \left(\sum_{i=0}^{r-1} \frac{1}{\lambda_i} \left(1 + \frac{S_i^2}{S} (\frac{d^{1-t} - d^{1-r}}{1-d}) + (\frac{S_i}{S} w)^2 \right) + \frac{1}{\lambda_r} \left(\frac{S_r^2}{S} (\frac{d^{1-t} - d^{1-r}}{1-d}) + (\frac{S_r}{S} w)^2 \right) \right) \mu^2 \\
&= \left(\sum_{i=0}^{r-1} \frac{1 + \frac{S_i^2}{S} (\frac{d^{1-t} - d^{1-r}}{1-d} + \frac{w^2}{S})}{1 + \frac{S_i}{S}(t - r + w)} + \frac{S_r (\frac{d^{1-t} - d^{1-r}}{1-d} + \frac{w^2}{S})}{t - r + w} \right) \mu \qquad (4.7)
\end{aligned}
$$

4.5 Unaligned d-heaps

The cache performance of a d-heap in which sets of siblings are not cache block aligned can be predicted with a simple change to the cache-aligned analysis. In the cache-aligned analysis, I know that examining a set of siblings will touch one cache block. In the unaligned case this is not necessarily true. A set of siblings uses de bytes of memory. On average, the chance that a set of siblings crosses a cache block boundary is $\frac{de}{B}$. In the event that the siblings do cross a cache block boundary, a second block will need to be touched. Thus on average, we expect $1 + \frac{de}{B}$ cache blocks to be touched when examining a set of siblings. This simple change yields the following new intensities

$$\lambda_{ij} = \begin{cases} (1 + \frac{de}{B})\mu & \text{if } 0 \leq j < r \text{ and } i = j, \\ \frac{S_i}{N_x}(1 + \frac{de}{B})\mu & \text{if } j = (x, y) \text{ and } r \leq x < t \text{ and } 0 \leq y < \frac{N_x}{S}, \\ \frac{S_i}{S} w\mu & \text{if } j = o, \\ 0 & \text{otherwise.} \end{cases}$$

$$\lambda_i = \begin{cases} (1 + \frac{de}{B})\mu + \frac{S_i}{S}(t - r)(1 + \frac{de}{B})\mu + \frac{S_i}{S} w\mu & \text{if } 0 \leq i < r, \\ \frac{S_r}{S}(t - r)(1 + \frac{de}{B})\mu + \frac{S_r}{S} w\mu & \text{if } i = r. \end{cases}$$

$$\lambda = t(1 + \frac{de}{B})\mu + w\mu$$

An expression for η can be derived by substituting these intensities into Equation 4.2 and reducing.

4.6 Validation

In order to validate these analyses, I compare the collective analysis predictions with the trace-driven simulation results for heaps running in the hold model from Chapter 3. To match the simulations, I set the cache size equal to 2 megabytes, the cache block size to 32 bytes and the heap element size to 4 bytes.

The quantity I compare is miss intensity ($\lambda - \eta$). Miss intensity is an interesting measure, as it predicts how many times an algorithm must service cache misses. I compare the model's predictions with the number of misses per iteration observed by the cache simulator. Figure 4.2 shows this comparison for a traditional heap and an aligned 2, 4 and 8-heap with $w = 0$ and a range of N between 1,000 and

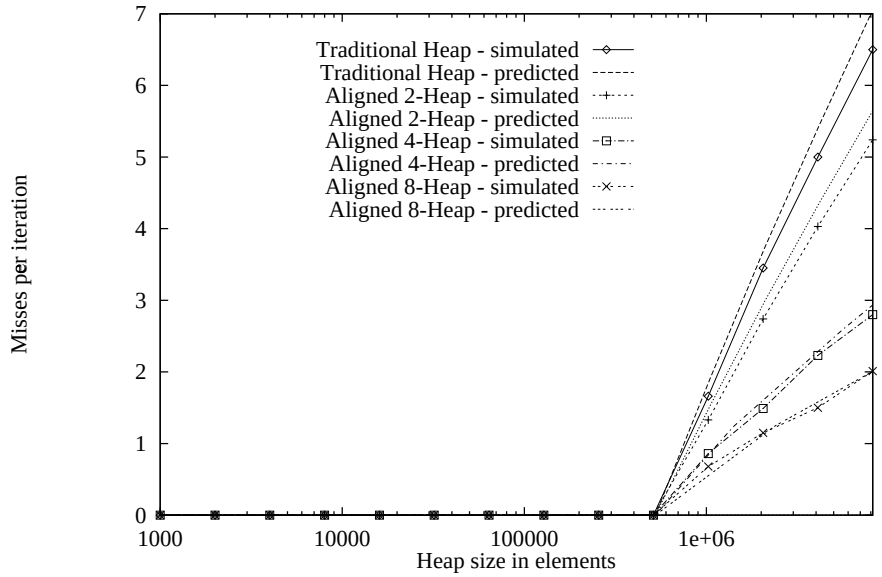

Figure 4.2: Collective analysis vs trace-driven simulation for heaps with no outside work. Simulated cache size is 2 megabytes, block size is 32 bytes and 4 elements fit per block.

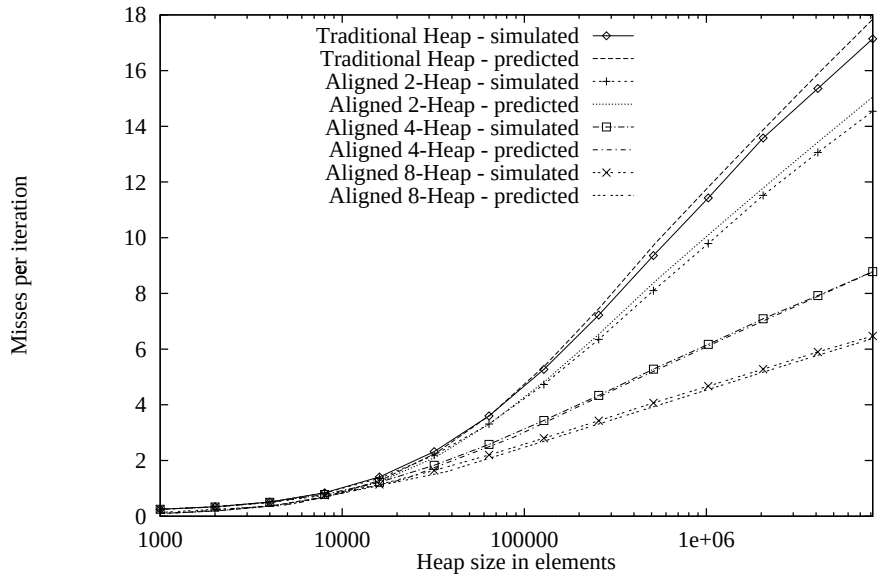

Figure 4.3: Collective analysis vs trace-driven simulation for heaps with outside work. Simulated cache size is 2 megabytes, block size is 32 bytes and 4 elements fit per block.

8,192,000. Since I do not consider startup costs and since no work is performed between the *remove-min* and the *add*, I do not predict or measure any cache misses until the size of the heap grows larger than the cache. This graph shows that the miss intensity predicted by the collective analysis closely matches the results of the cache simulations. This indicates that while the heap may not be composed of independent stochastic processes, they can be used to accurately model the heap's memory system behavior.

I next compare the model predictions against simulations for a system in which 25 random accesses are performed each iteration. Figure 4.3 shows the miss intensities for a traditional heap and an aligned 2, 4 and 8-heap with $w = 25$ and a range of N between 1,000 and 8,192,000. Again, we see that the predictions of the collective analysis closely match the simulation results. It is interesting to see that collective analysis overpredicts the misses incurred by the binary heaps and underpredicts for the 8-heap. This is partly due to the assumption that sets of siblings sharing a cache block are not loaded together. Since 8 heap elements fit in a cache block, this assumption is always true for the 8-heap and no inaccuracy is introduced in this case. For the binary heaps, however, collective analysis underestimates the number of heap elements brought into the cache, and this causes a slight overprediction of cache misses.

4.7 Hot Structure

A common problem is that a particular application will perform well 90% of the time but will occasionally perform poorly. These occasional slowdowns are often due to variations in the relative placement of dynamically allocated blocks of memory. A bad allocation may result in two heavily accessed blocks of memory conflicting in the cache resulting in a large number of cache misses. In this section, I use collective analysis to investigate the impact that the relative position of memory blocks has on cache performance. I extend the cache-aligned analysis to include a small *hot structure* to which a large number of accesses are made. This hot structure is intended to model a small heavily accessed memory object such as a square root lookup table. By varying the placement of this hot structure relative to the other memory blocks, I model both the best and worst case cache performance of the system.

In the cache-aligned analysis, outside work consists of w random accesses to a

cache sized array. I also augment this outside work to include w_h random accesses to a second array of size S_h. I now perform two collective analyses, one in which this second array is positioned to minimize cache misses, and a second in which the array is positioned to maximize cache misses.

4.7.1 Best Case

The area of the cache least accessed is cache region r which represents the cache blocks that remain after all of the heap-level sized regions have been allocated (see Figure 4.1). For simplicity, I assume that the hot structure fits in region r, that is $S_h \leq S_r$. To accommodate the hot structure, I divide region r into two regions, r and $r+1$. The new region r is the old region r with the hot structure taken out, and has size $S - \frac{d^r-1}{d-1} - S_h$. Region $r+1$ holds the hot structure and is of size S_h. To model the accesses to the hot structure, I add a new process called o_h. Only one change needs to be made to the access intensities from Section 4.4 to model the best case cache performance with the hot structure; a new expression needs to be added to indicate that process o_h accesses region $r+1$ with intensity $w_h\mu$. The resulting access intensities are:

$$\lambda_{ij} = \begin{cases} \mu & \text{if } 0 \leq j < r \text{ and } i = j, \\ \frac{S_i}{N_x}\mu & \text{if } j = (x, y) \text{ and } r \leq x < t \text{ and } 0 \leq y < \frac{N_x}{S}, \\ \frac{S_i}{S}w\mu & \text{if } j = o, \\ w_h\mu & \text{if } j = o_h \text{ and } i = r+1, \\ 0 & \text{otherwise.} \end{cases}$$

Values for λ and η can be derived by plugging these intensities into equations 4.1 and 4.2 and reducing.

4.7.2 Worst Case

The most heavily accessed cache blocks are those that hold the first few levels of the heap. The block in cache region 0 is accessed the most often, the blocks in region

1 the second most often, and so on. For the worst case cache performance, the hot structure is placed in the cache so that it conflicts with the top levels of the heap. I assume that the hot structure starts at the beginning of region 0, and for simplicity I assume that it ends at the end of region r_h. That is, I assume that $S_h = \sum_{i=0}^{r_h} S_i$. I again add a new process o_h to model the w_h accesses made to the hot structure every iteration. The only change that needs to be made to the access intensities from Section 4.4 is to add an expression indicating that regions $0 \ldots r_h$ are accessed uniformly by process o_h for a total intensity of $w_h\mu$. The new access intensities are:

$$\lambda_{ij} = \begin{cases} \mu & \text{if } 0 \leq j < r \text{ and } i = j, \\ \frac{S_i}{N_x}\mu & \text{if } j = (x, y) \text{ and } r \leq x < t \text{ and } 0 \leq y < \frac{N_x}{S}, \\ \frac{S_i}{S}w\mu & \text{if } j = o, \\ \frac{S_i}{S_h}w_h\mu & \text{if } j = o_h \text{ and } 0 \leq i \leq r_h, \\ 0 & \text{otherwise.} \end{cases}$$

Values for λ and η can be derived by plugging the intensities into 4.1 and 4.2 and reducing.

4.7.3 Comparison

I now compare the predictions of the best and worst case analysis of the hot structure and the original analysis with no hot structure. For the comparison, I use an 8-heap with 4 byte heap elements and a total of 25 outside accesses per iteration. For the scenario with no hot structure, I use the analysis from Section 4.4 and set $w = 25$ to indicate that all of the outside work takes place in the cache sized array. For both of the scenarios with a hot structure, I set the size of the hot structure to 73 ($S_h = 73$, $r_h = 2$). I divide the 25 accesses so that 15 are made to the cache sized array and 10 to the hot structure ($w = 15$ and $w_h = 10$).

To validate the analyses, the predictions for all three scenarios are compared with trace driven simulation results. For the best and worst case scenarios, the implementation being simulated was configured to explicitly place the hot structure

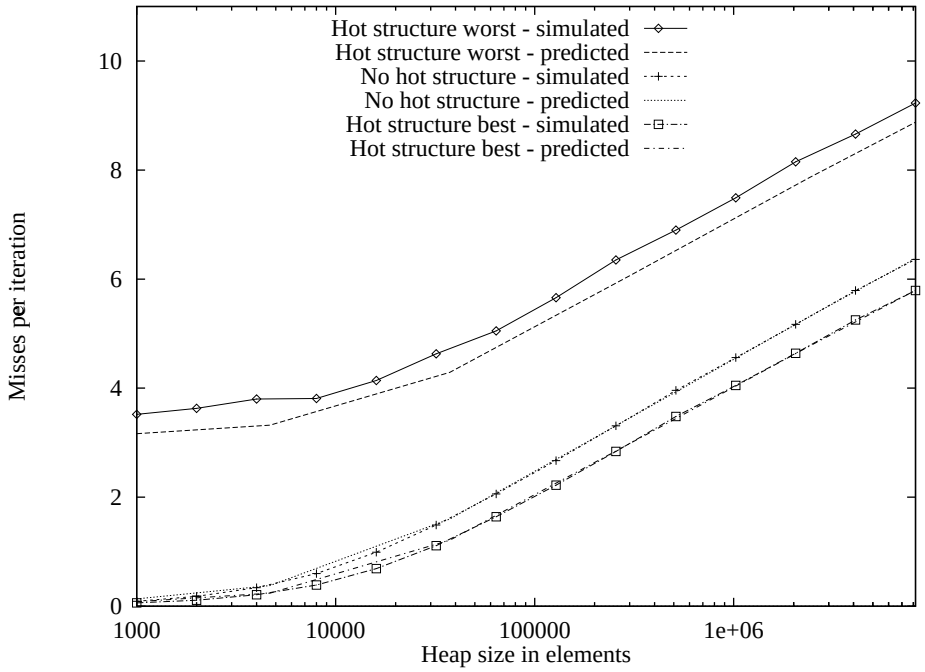

Figure 4.4: The effects of relative block placement on cache performance. Simulated cache size is 2 megabytes, block size is 32 bytes and 8 elements fit per block.

---

so that it conflicted either with region r or with regions $0 \ldots r_h$ as the analysis assumes. Figure 4.4 shows the cache misses per iteration for both predictions and simulations varying heap size from 1,000 to 8,192,000.

The predictions and simulations in this graph largely agree. The best case hot structure and no hot structure scenarios are predicted very accurately. The predictions for the worst case hot structure vary from the simulation results by a constant fraction of a cache miss throughout the range of heap sizes. Despite the differences, both the predictions and simulations indicate that the placement of the hot structure has a large impact on cache performance. With a 1,000 element heap, both the best case scenario and the no hot structure scenarios incur almost no cache misses. In contrast, the worst case scenario incurs more than three misses per iteration due to the interaction between the hot structure and the heap. This is a large difference considering that only ten memory references are made to the hot structure per iteration. As the heap grows larger, the cache misses per iteration grow for all three scenarios,

but the worst case scenario remains more than three misses higher than the others due to the interference from the hot structure.

As I have shown, the relative placement of memory blocks can have a significant impact on the cache performance of algorithms, and collective analysis is one of the first analytical tools that can be used to study this effect. This example shows how easily collective analyses can be incrementally extended. Both the best and worst case analyses required very simple changes to the base analysis. This makes collective analysis good for answering "what if?" questions about algorithm structure and optimizations.

4.8 Applicability

While the predictions for the d-heap are accurate, collective analysis is limited as a general tool for predicting cache performance. The largest drawback of collective analysis is the assumption that memory accesses are uniformly distributed within cache regions. While this assumption accurately models the memory behavior of heaps, it does not apply to the vast majority of algorithms. Collective analysis would have to be extended to model other reference patterns for it to be generally applicable.

Another drawback of collective analysis is that it assumes that the relative location of a memory object in the virtual address space is known at analysis time. If an algorithm dynamically allocates memory, the new object's location relative to other objects will not be known, limiting the applicability of this technique.

The real strength of collective analysis is the intuition it provides, and the process of performing the analysis is just as valuable as the predictions it makes. As evidence of this, I cite the two heap optimizations presented in Chapter 3. I first used collective analysis to investigate the impact of larger fanouts. After performing the analysis, the predictions did not match the cache simulation results. Investigation revealed that the collective analysis made the assumption that sets of siblings were cache aligned, and an examination of the heap implementation revealed that they were not. Padding the heap array in the implementation made the simulation results agree with the analysis predictions. Performing the collective analysis revealed an inefficiency that I had not previously considered.

4.9 Summary

This chapter introduces collective analysis, a purely analytical technique that can be used to examine the cache performance of algorithms without implementations and without address traces. Collective analysis provides a framework within which the memory behavior of an algorithm is approximated, and this approximation is used to predict cache performance for direct mapped caches. The main drawback of collective analysis is that it makes assumptions regarding the distribution of memory accesses that are not true for most algorithms. Collective analysis has the advantages that the model is easy to understand, the analyses are easy to incrementally extend and it allows fast predictions to be made for varying cache and algorithm configurations.

In the chapter, collective analysis is performed on both cache-aligned and unaligned d-heaps and the accuracy of the analysis is verified by comparing the predictions with trace-drive cache simulation results. Collective analysis is also used to explore the impact that the placement of dynamically allocated memory blocks can have on cache performance.

Chapter 5

A Comparison of Priority Queues

While two algorithms may solve the same problem, their instruction mixes and memory access patterns may vary widely. The increase in cache miss penalties can change the relative cost of instructions, which in turn has the potential to change the relative performance of algorithms. For this reason, it is important to periodically reexamine the performance of different algorithms for common tasks. The danger of not periodically investigating the impact that shifts in technology have on relative performance is that the results of outdated studies may be applied to architectures for which they are not relevant.

Such is the case with Jones's 1986 study of the performance of priority queues [Jones 86]. The study was thorough, comparing many different algorithms and data distributions, and is commonly cited as evidence that one priority queue algorithm outperforms another. In this chapter, I investigate how increases in cache miss penalties have affected the relative performance of priority queues by reproducing a subset of Jones's experiments. I examine the performance of implicit heaps, skew heaps and splay trees in the hold model as Jones did in his study. The results of Jones's experiments indicate that for the architectures of that generation, pointer-based self-balancing priority queues such as splay trees and skew heaps perform better than the simple implicit heaps. In my experiments, however, the pointer-based queues perform poorly due to poor memory system behavior. The pointers used to represent the queue's structure drastically increase the queue element size causing fewer elements to fit in the cache. The result is that as queue size is increased, cache misses occur sooner and the cache miss curve climbs more steeply than the curves for non-pointer based queues. In contrast to Jones's results, the heaps from Chapter 3 outperform

the pointer-based queues by up to a factor of four in my experiments. Their implicit structure results in a compact representation with low memory overhead and this far outweighs any instruction cost benefits that self-balancing techniques afford the other queues.

5.1 The Experiment

I examine the performance of five priority queue implementations operating in the hold model. In my experiments I compare a traditional heap, a cache-aligned 4-heap, a top-down skew heap and both a top-down and a bottom-up splay tree. The implementations of the traditional heap and the aligned 4-heap are described in Chapter 3. The implementations of the skew heap and the bottom-up splay tree are taken from an archive of the code used in Jones's study. The top-down splay tree implementation is an adaption of Sleator and Tarjan's code [Sleator & Tarjan 85]. As Jones did in his study, the queues are run in the hold model, and no work is performed between the *remove-min* and the *add* each iteration. In my experiments, a queue element consists of an 8 byte key and no data. This varies slightly from Jones's experiments where he used 4 byte keys. A larger key value was chosen to allow extended runs without the keys overflowing.

The priority queues are initially seeded with exponentially distributed keys, and the priority queues are run in the hold model for 3,000,000 iterations to allow the queue to reach steady state. The performance of the queue is then measured for 200,000 iterations. Executions are run on a DEC Alphastation 250. Cache simulations are configured for a 2 megabyte direct-mapped cache and a 32 byte block size.

In my experiments, the queue size is varied from 1,000 to 1,024,000 elements, although the bottom-up splay tree is only run up to a size of 512,000 elements due to physical memory pressure. This differs from Jones's study where he used considerably smaller queues ranging from size 1 to 11,000.

5.2 Results

Figure 5.1 shows a graph of the dynamic instruction count of the five priority queues. As we expect, the number of instructions executed per iteration grows with the logarithm of the number of elements for all five of the priority queues. The 4-heap executes

fewer instructions than the traditional heap as predicted by the analysis in Chapter 3. The bottom-up splay tree executes almost exactly the same number of instructions as the traditional heap. For comparison Figure 5.2 shows a graph of the execution times observed by Jones on a Vax 11/780 in his original study for a traditional heap, a top-down skew heap and a bottom-up splay tree. This graph suggests that on a Vax the splay-trees execute fewer instructions than the traditional heap, while my executions indicate that on today's machines they do not. This difference can be attributed to the fact that Jones's experiments were run on CISC machines with memory-memory instructions while mine were run on a load-store RISC machine. Splay trees are memory intensive, executing almost three times as many loads and stores as the heap. The pointer manipulations performed by the self-balancing queues are fairly inexpensive on the architectures that Jones used. On load-store RISC machines, however, these pointer manipulations translate into multiple instructions, increasing the instruction cost of the pointer-based queues relative to the implicit heaps.

The top-down splay tree performed the most instructions by a wide margin. To be fair, I must say that while the top-down splay trees were coded for efficiency, they did not receive the heavy optimization that Jones gave his codes or that I gave the heaps.

The cache performance of the priority queues is compared in Figure 5.3. As the experiments were run in the hold model with no work between iterations, cache misses do not occur after warmup unless the queue is larger than the cache. It is at this point that an important difference in the priority queues is revealed. Depending on their algorithms for adding and removing elements, the queues have different sized elements and this has a huge impact on cache performance. Adding pointers to a queue element increases its size reducing the number of elements that fit in the cache which in turn reduces locality.

The implicit heap elements require no pointers and the elements only contain the keys. The top-down skew heap and top-down splay tree, on the other hand, both require a left and right pointer per element, adding 16 bytes of overhead to each queue element. In addition, queue elements for the pointer-based queues are allocated from the system memory pool, and 8 bytes of overhead are incurred each time an element is allocated[1]. The pointer overhead combined with the overhead of the system memory

[1] There are memory pool implementations that do not incur per-allocation overhead for small

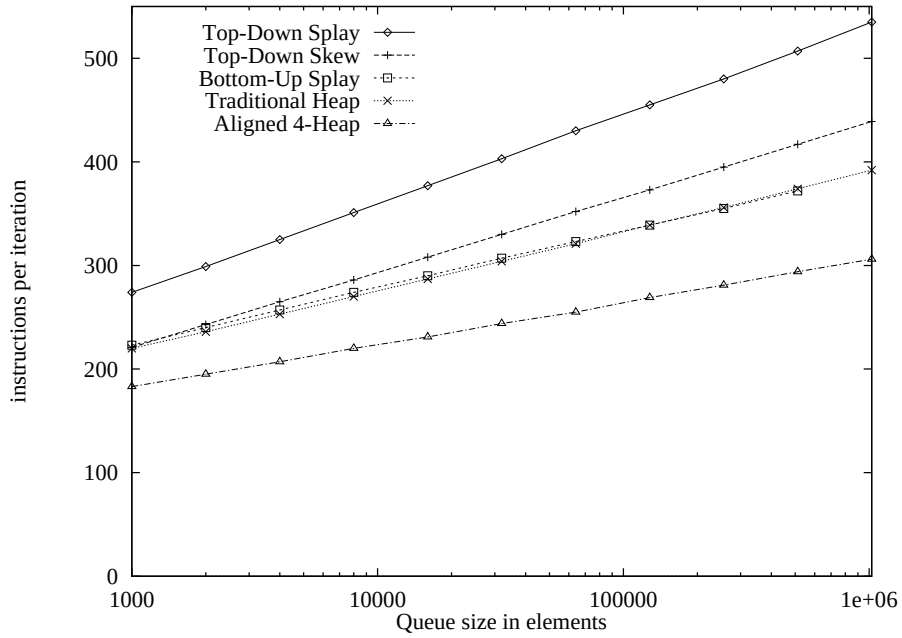

Figure 5.1: Instruction counts for five priority queues in the hold model.

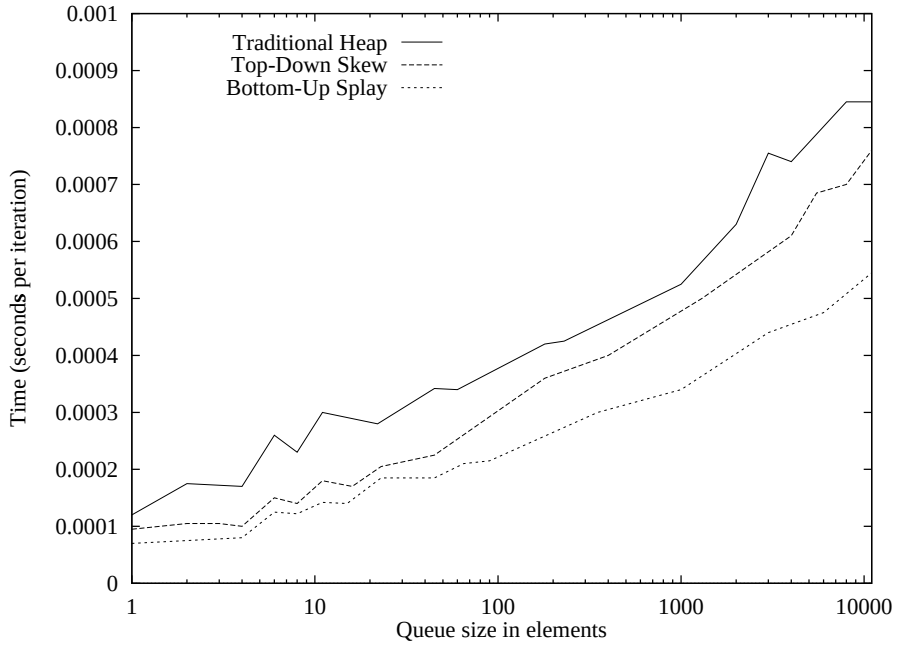

Figure 5.2: Execution time for three priority queues on a VAX 11/780 operating in the hold model. These graphs show the priority-queue performance that Jones observed in his original study.

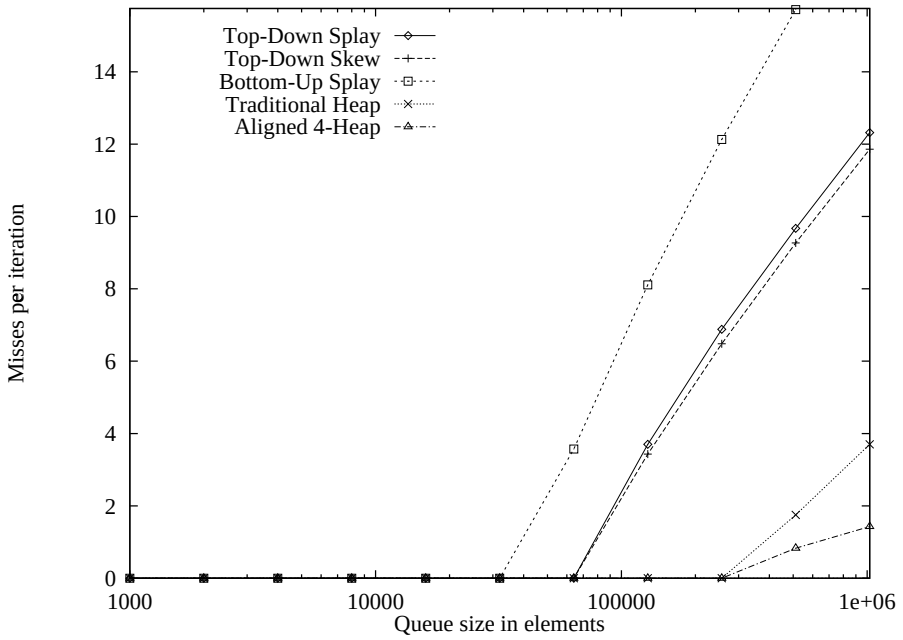

Figure 5.3: Cache performance of five priority queues in the hold model. Simulated cache size is 2 megabytes, block size is 32 bytes and 4 elements fit per block.

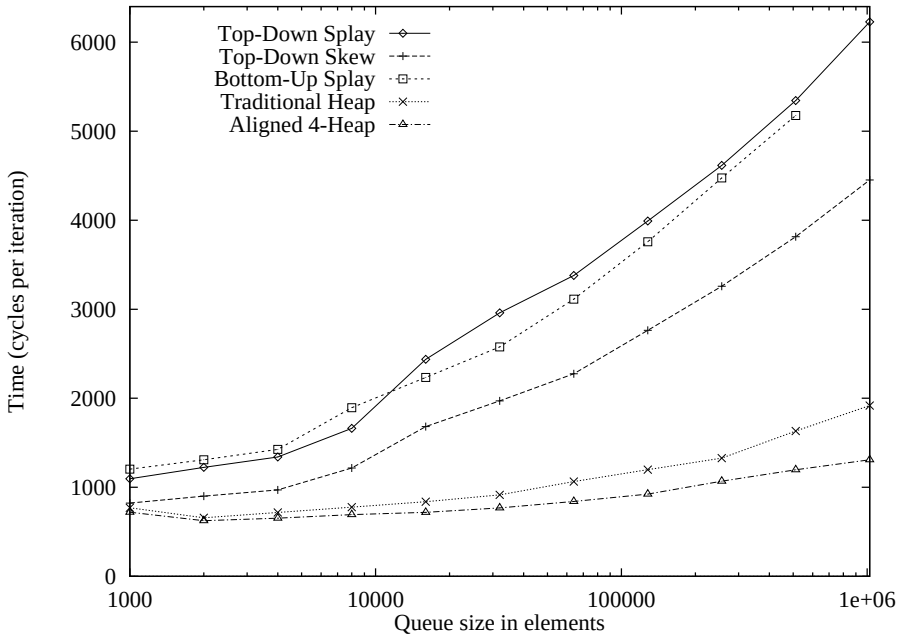

Figure 5.4: Execution time for five priority queues in the hold model. Executions on a DEC Alphastation 250 with 4 elements per block.

pool results in an element size of 8 (key) + 8 (left) + 8 (right) + 8 (memory pool) = 32 bytes, four times larger than a heap element. As a result, the top-down skew heaps and top-down splay trees start incurring cache misses at one-quarter the queue size of the heaps. The queue requiring the most overhead is the bottom-up splay tree with three pointers per element (left, right and parent). The result is that the splay tree has the largest footprint in memory and is the first to incur cache misses as the queue size is increased.

The execution times for the five queues executing on an Alphastation 250 are shown in Figure 5.4. Due to their low instruction cost and small cache miss count, the aligned 4-heap performs best, with the traditional heap finishing second. This varies from Jones's findings in which the implicit heaps finished worse than splay trees and skew heaps. The two splay tree implementations have similar execution times in my experiments, with the higher instruction count for the top-down splay tree offsetting the higher cache miss count for the bottom-up splay tree. Despite the locality benefits splay trees afford, their large footprint in memory causes them to perform poorly in my study.

5.3 Impressions

These experiments illustrate the importance of a design that is conscious of memory overhead and the effects of caching, and strongly suggest that future designs will need to pay close attention to memory performance if good overall performance is to be achieved. The goal of this chapter is not to convince the reader that heaps are the best priority queue. There are obvious optimizations that could be applied to the splay trees and skew heaps which would reduce their cache misses and improve their performance. One good starting point would be to allocate queue nodes in large bundles to minimize the overhead incurred in the system memory pool. By allocating elements 1000 at a time, overhead could be reduced from 8 bytes per queue element to 8 bytes per 1000 queue elements.

An additional optimization would be to store queue nodes in an array and represent references to the left and right elements as offsets in the array rather than using 64 bit pointers. If the queue size was limited to 2^{32}, the 8 byte pointers could

objects [Grunwald et al. 93]. In OSF/1, however, the default `malloc` incurs an 8 byte overhead per allocation.

be replaced with 4 byte integers instead. These two optimizations alone would reduce the total overhead for the top-down skew heap to 4 (left) + 4 (right) = 8 bytes rather than 8 (left) + 8 (right) + 8 (memory pool) = 24 bytes. These simple changes would have a significant impact on the performance of all of the pointer-based priority queues used in this study. It is unlikely, however, that the splay trees or skew heaps could be made to perform as well as the implicit heaps for the applications I looked at, since the heaps have such low instruction counts and have no memory overhead due to their implicit structure.

5.4 Summary

This chapter revisits Jones's study of the performance of priority queues. Implicit heaps, skew heaps and splay trees are run in the hold model and their relative performance is compared with the results from Jones's study. Jones's results indicated that the pointer-based self-balancing queues such as splay trees outperformed the simpler queues. My results, however, show that the high memory overhead of the pointer-based self-balancing queues results in poor memory behavior which in turn translates in bad overall performance on today's machines. As well as demonstrating the benefits of cache conscious design, this chapter shows the importance of periodically examining the impact that shifts in technology have on the relative performance of common algorithms.

Chapter 6

Sorting Efficiently in the Cache

One of the most common tasks computers perform is sorting a set of unordered keys. Sorting is a fundamental task and hundreds of sorting algorithms have been developed. This chapter explores the potential performance gains that cache-conscious design offers in the context of three popular comparison-based sorting algorithms: heapsort [Williams 64], mergesort [Holberton 52][1] and quicksort [Hoare 62].

For each of these three sorting algorithms, I choose an implementation variant with potential for good overall performance and then heavily optimize this variant using traditional techniques. Using each optimized algorithm as a base, I then develop and apply memory optimizations in order to improve cache performance and overall performance. Trace-driven simulations and actual executions are used to measure the impact the memory optimizations have on performance. For all three sorting algorithms, cache misses were significantly reduced: up to 57% for heapsort, 90% for mergesort and 40% for quicksort. These reductions in cache misses translate into speedups of 82% for heapsort, 75% for mergesort and 4% for quicksort. I discuss the optimization and performance of heapsort, mergesort and quicksort in Sections 6.1, 6.2 and 6.3 respectively, and I considered the performance of the three collectively in Section 6.4.

---

[1] This reference discusses the first computer implementations of mergesort. Mergesort was first implemented in card sorting machines in the 1930s [Knuth 73].

6.1 Heapsort

I first analyze the heapsort algorithm and examine how cache-conscious design improves performance. The heapsort algorithm first builds a heap containing all of the keys and then removes them all from the heap in sorted order [Williams 64]. With n keys, building the heap takes $O(n \log n)$ steps, and removing them in sorted order takes $O(n \log n)$ steps. In 1965 Floyd proposed an improved technique for building a heap with better average case performance and a worst case of $O(n)$ steps [Floyd 64]. The standard algorithms for adding and removing elements from a heap as well as Floyd's method for building a heap are discussed in Chapter 3.

6.1.1 Base Algorithm

Before applying any memory optimizations, I develop an otherwise well optimized version of the algorithm. As a base heapsort algorithm, I follow the recommendations of algorithm textbooks and use a binary heap constructed using Floyd's method. This algorithm spends the majority of its time percolating elements down the heap. An inefficiency in this process is that a parent may have either zero, one or two children. This requires that two checks be made per heap level: one to see if an element has any children, and another to see if it has only one child. This can be reduced to one check if parents are guaranteed to have either zero or two children. This can be guaranteed by ensuring that a maximal key is always placed after the last heap element in the array. In this way, parents with only one child will appear to have two, a normal child and a maximal child that will never be chosen as the minimum. The effect of this change is that an extra compare may be performed at the bottom layer of the heap, but all other layers need only perform one check to see if they have children. This is the only non-memory optimization applied to the base heapsort. The literature contains a number of optimizations that reduce the number of comparisons performed for both adds and removes [De Graffe & Kosters 92, Carlsson 91, Gonnet & Munro 86], but in practice these do not improve performance and I do not include them in the base heapsort.

6.1.2 Memory Optimizations

To this base algorithm, I now apply memory optimizations in order to further improve performance. Simulations from Chapter 3 show that William's simple algorithm for building a heap incurs fewer cache misses than Floyd's method. Chapter 3 also develops two optimizations for reducing the number of cache misses incurred by the *remove-min* operation. I use all of these optimizations to construct a memory-tuned heapsort. The memory-tuned heapsort makes use of the architectural constants to choose a heap fanout such that a set of siblings fills a cache block. The algorithm dynamically chooses between the Repeated-Adds method and Floyd's method for building a heap. If the heap is larger than the cache and Repeated-Adds can offer a reduction in cache misses, it is chosen over Floyd's method. The root of the heap is then aligned so that sets of siblings do not cross cache block boundaries, the heap is built with the chosen method, and keys are removed from the heap using the memory optimized *Remove-Min* until the sorted set is complete.

6.1.3 Performance

I now compare the performance of the base heapsort and the memory-tuned heapsort. The performance of each algorithm is measured by sorting sets of 64 bit uniformly distributed integers. Figure 6.1 shows a graph of the dynamic instruction count of the two heapsort algorithms varying the set size from 1,000 to 4,096,000 keys. With 64 bit keys, 4 heap elements fit per cache block and the memory tuned heapsort chooses a 4-heap. As a result, the memory-tuned heapsort executes fewer instructions than the base heapsort. Figure 6.2 shows the number of cache misses incurred per key for a simulated 2 megabyte cache with a 32 byte block size. For sets which do not fit in the cache, the minimum 64 bits / 32 bytes = 0.25 compulsory misses are incurred per key for both algorithms. For sets larger than the cache, the memory-tuned heapsort incurs fewer than half of the misses of the base heapsort.

Up to this point, all of the cache simulations have been run with two megabyte caches, the same size as the second level cache of the DEC Alphastation 250. I have focused on second level cache performance because second level caches have high miss penalties and significantly affect overall performance. While first level miss penalties are small, on the order of five cycles, they also have the potential to impact overall performance. Figure 6.3 shows the number of cache misses per key for the heapsort

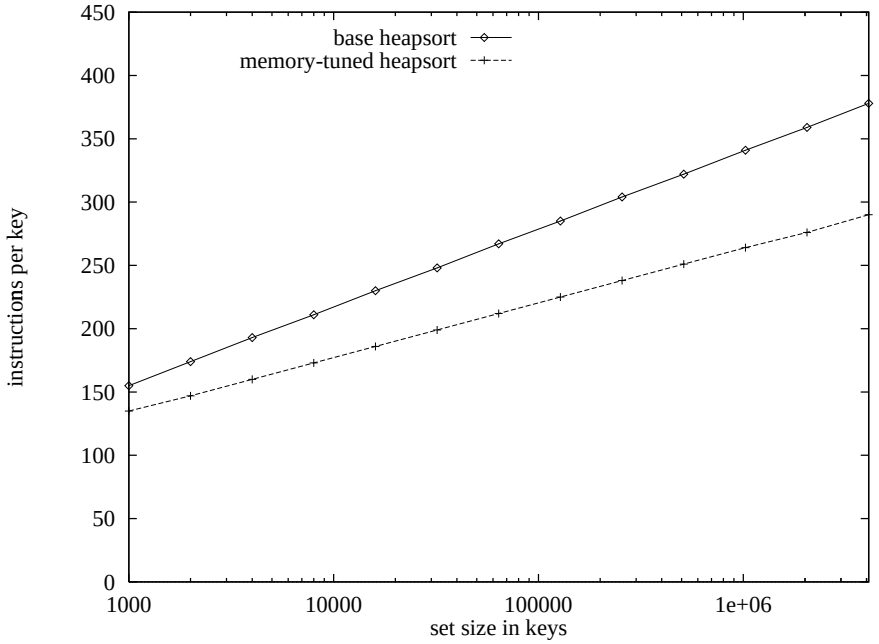

Figure 6.1: Instruction counts for heapsort.

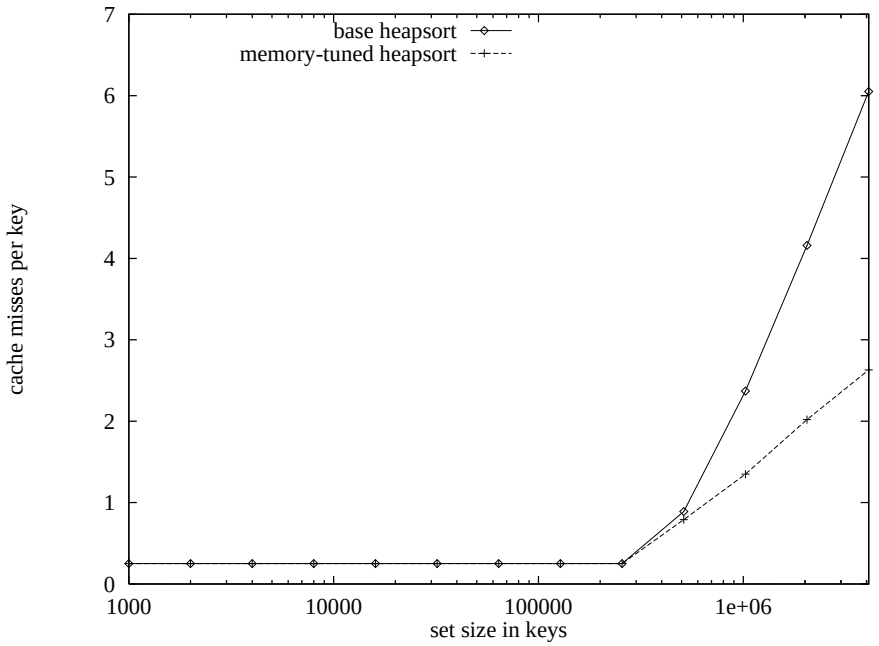

Figure 6.2: Cache performance of heapsort. Simulated cache size is 2 megabytes, block size is 32 bytes.

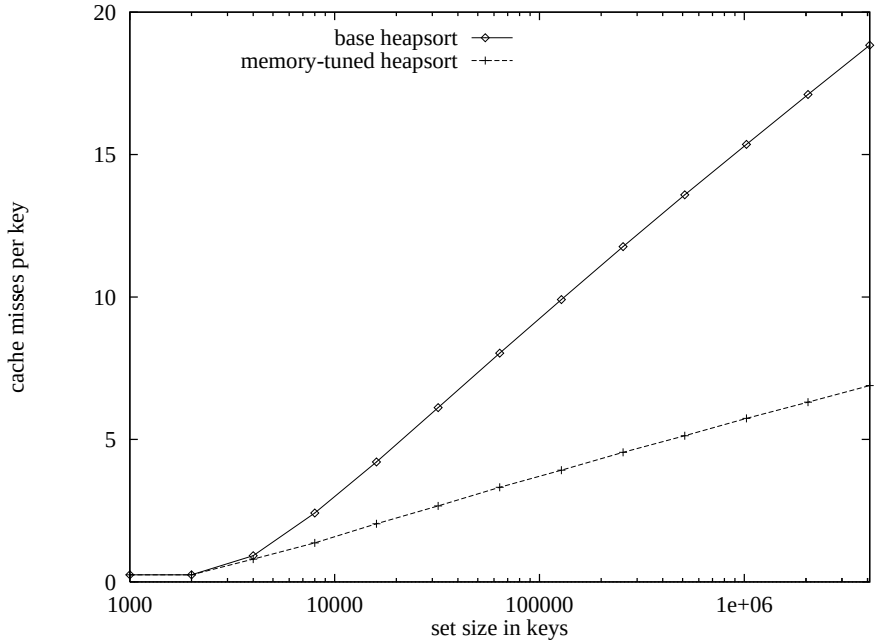

Figure 6.3: First level cache performance of heapsort. Simulated cache size is 16 kilobytes, block size is 32 bytes.

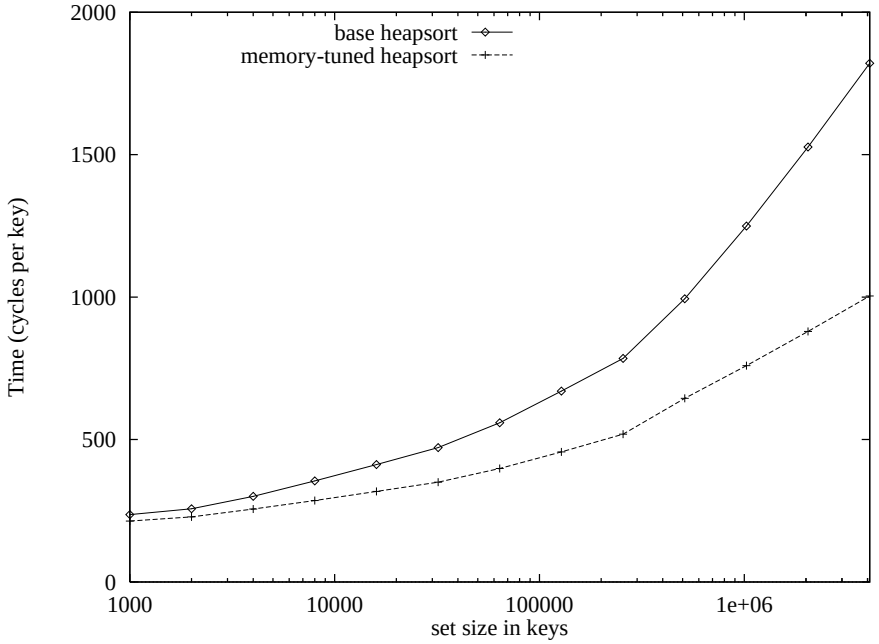

Figure 6.4: Execution time for heapsort on a DEC Alphastation 250.

algorithms with a 16 kilobyte direct-mapped cache with a 32 byte block size, the same as the first level cache of the DEC Alphastation 250. In this graph, both algorithms incur non-compulsory misses far earlier due to the smaller cache size. Since the optimizations of the *remove-min* operation take into account only the cache's block size, improvements in memory performance are not dependent on cache capacity. As a result, the memory-tuned heapsort incurs fewer than half the misses of the base heapsort, similar to the simulation results for the second level cache.

Finally, Figure 6.4 shows the execution time for the two heapsort algorithms on a DEC Alphastation 250. The memory-tuned heapsort initially outperforms the base heapsort due to lower instruction cost, and as the set size is increased the gap widens due to differences in first level cache misses. When the set size reaches the size of the second level cache (262,144 keys), the curves steepen sharply due to the high cost of second level cache misses. For 4,096,000 keys, the memory-tuned heapsort sorts 81% faster than the base heapsort.

6.2 Mergesort

I now perform a similar analysis and optimization of mergesort. Two sorted lists can be merged into a single sorted list by traversing the two lists at the same time in sorted order, repeatedly adding the smaller key to the single sorted list. By treating a set of unordered keys as a set of sorted lists of length one, the keys can be repeatedly merged together until a single sorted set of keys remains. Algorithms which sort in this manner are known as mergesort algorithms, and there are both recursive and iterative variants [Holberton 52, Knuth 73]. Recursive mergesort is a classic divide-and-conquer algorithm which recursively sorts the left and right half of the set and then merges the sorted halves together. Iterative mergesort uses the same merging process but merges the lists in a different order. It first walks through the set of keys merging the sorted lists of length one into sorted lists of length two. Then it merges the lists of length two into lists of length four. Successive passes through the set of keys create longer and longer sorted lists until there is one sorted list holding all of the keys.

For a base algorithm, I chose an iterative mergesort since it is as easy to implement as recursive mergesort and has a lower instruction cost. Iterative mergesort was also chosen because it is very amenable to traditional optimization techniques, allowing me

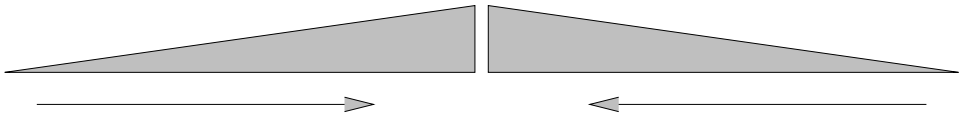

Figure 6.5: The layout of pairs of lists in the base mergesort.

to create a mergesort that performs extremely well according to traditional metrics.

6.2.1 Base Algorithm

The iterative mergesort described by Knuth is chosen as the base algorithm [Knuth 73, Algorithm N, Pg. 162]. This base mergesort operates on an array of keys and uses an auxiliary array to store partially merged lists. While there are algorithms for merging two sorted lists in place in linear time [Huang & Langston 88], these algorithms are extremely complex and perform poorly in practice. The first iteration in Knuth's mergesort walks through the source array, writing merged lists of length two into the auxiliary array. The lists in the auxiliary array are then merged into lists of length four and are written back into the source array. By keeping track of which array is the source and which is the destination, the lists can be merged back and forth between arrays without copying. A check is performed at the end, and if the sorted set finished up in the auxiliary array it is copied back into the source array.

An inefficiency of Knuth's array-based mergesort is the need to check a number of end conditions during each merge step. At the start of a merge, the algorithm knows the number of keys in each of the two sorted lists and therefore the number of keys in the final list. The algorithm does not know, however, how fast each list will be consumed and which list will be exhausted first. In an extreme case, all of the keys in one list will be consumed before any keys are consumed from the other list. For this reason, individual merge steps must check to see if there are keys remaining in each of the lists. Since these checks reside in the innermost loop of the algorithm, they represent an important inefficiency. Sedgewick shows that by reversing the order of the second list and merging from the outside to the middle, two lists can be merged without checking to see if lists have been exhausted (see Figure 6.5). When one of the two lists is fully consumed, its list index will be incremented off the end of the

list and will point at the largest key in the other list. Since this key is larger than each of the other keys in the remaining list, it serves as a sentinel and will not be chosen until the final merge step. This optimization requires changing the merge algorithm so that it can create both forward and reverse sorted lists. An iteration now merges pairs of lists and alternately creates forward and reverse sorted lists. This optimization complicates the algorithm but provides a big reduction in instruction cost as it eliminates instructions from the innermost loop.

A common optimization of many sorting algorithms is to fully sort small subsets of keys with a simple n^2 sorting algorithm. When the set size is small, simple sorting algorithms such as insertion sort can sort faster than $n \log n$ sorts due to smaller constants. Sedgewick recommends this for quicksort [Sedgewick 78], and I adapt the idea to mergesort. I make an initial pass through the source array creating sorted lists of length four with an efficient inline sort and then use mergesort to complete the sorting.

The final non-memory optimization I apply to mergesort is to unroll the innermost loop eight times [Lowney et al. 93]. Unrolling the loop has two benefits: it allows the loop bounds check to be amortized over eight merge steps rather than one, as well as creating a much larger block of code which allows the compiler's instruction scheduler to make better scheduling decisions. My base mergesort algorithm incorporates all of these optimizations and has very low instruction cost, executing fewer than half as many instructions as the base heapsort.

6.2.2 Memory Optimizations

Despite my efforts to produce an excellent base algorithm, an examination of the base mergesort's memory performance reveals several other optimization opportunities. The first issue to consider is the placement of the auxiliary array relative to the source array. In Chapter 4, we saw that relative block placement can have a significant impact on cache performance. Since mergesort is not an in-place algorithm and requires additional space equal to the size of the set of keys, the largest sized set we can sort within the cache is $CacheSize/2$. Half of the cache will hold the source array, and the other half will hold the auxiliary array. In order to avoid conflicts in the cache, these two arrays should be positioned so that they start $CacheSize/2$ apart in the cache. In all of the mergesort implementations, including the base mergesort,

the auxiliary array is positioned in memory so that the first key in the auxiliary array maps to a cache location $CacheSize/2$ away from the first key of the source array. This guarantees that any contiguous piece of the source array of size $CacheSize/2$ or smaller will not conflict with its counterpart in the auxiliary array.

The memory behavior of the iterative mergesort is far simpler to understand than the memory behavior of heapsort. Each pass sequentially walks through areas of the source array while sequentially writing into the destination array. This reference pattern results in good spatial locality. Blocks from the source array that are brought into the cache are sequentially traversed and all of the keys in the block are used. Similarly, the sequential writes to the destination array use all of the keys in a cache block and exhibit good spatial locality.

Unfortunately, the base mergesort algorithm has the potential for terrible temporal locality. Mergesort uses each data item only once per pass, and if a pass is large enough to wrap around in the cache, keys will be ejected before they are used again. If the set of keys is of size $CacheSize/2$ or smaller, the entire sort can be performed in the cache and only compulsory misses will be incurred. When the set size is larger than $CacheSize/2$, however, temporal reuse drops off sharply and when the set size is larger than the cache, no temporal reuse occurs at all. This inefficiency can be reduced by making passes small enough that temporal reuse can occur. This can be achieved by first breaking the set into subsets of size $CacheSize/2$ and fully sorting these subsets. Afterwards, these subsets can be formed into a single sorted set using standard merge passes. The initial sets of passes to create sorted lists of size $CacheSize/2$ will incur only one miss per block for both the source and auxiliary arrays. This locality improving optimization is called *tiling* and can be applied to simple loops by compilers [Wolfe 89]. Tiling the base mergesort drastically reduces the misses it incurs, and the added loop overhead increases instruction cost very little. I call the tiled version of the base mergesort *tiled mergesort.*

The tiled mergesort goes through two phases. The first phase fully sorts the half-cache sized pieces, and the second phase merges these pieces into a single set. The first phase has good cache behavior, but the second phase still suffers from the same problem as the base mergesort. Each pass through the source array in the second phase needs to fault in all of the blocks, and no reuse is achieved across passes if the set size is larger than the cache. To fix this inefficiency in the second phase, I replace

the $\lceil \log_2 n/(CacheSize/2) \rceil$ merge passes with a single pass that merges all of the pieces together at once. Multi-way merging is commonly used for external sorting, and Knuth devotes a section of his book to techniques for multimerging [Knuth 73, Sec. 5.4.1]. Since passes in the second phase exhibit no temporal reuse, it makes sense from a cache perspective to minimize the number of passes.

A k-way merge takes as input k sorted lists, and produces a single sorted list while reading through each of the k lists only once. A merge step in the multimerge examines the heads of the k lists to find the minimum key and and moves it to the output list. The multimerge repeatedly executes this merge step until all of the k lists are empty. If k is small, a linear search through the heads of the k lists can be used to find the minimum key. As k grows large, this is inefficient, and a common technique is to use a priority queue to hold the heads of the lists. When using a priority queue, a merge step in the multimerge removes the minimum key from the queue, and adds it to the output list. It then removes the head from list that the minimum key came from and adds it to the priority queue. Multimerge using a priority queue has the potential for good cache performance. Each key in the source and destination array will be accessed only once, and since the priority queue is small, it will not incur many cache misses. Unfortunately, if implemented as just described, multimerge can suffer from *thrashing*, a problem that occurs when two cache blocks that map to the same location are alternately accessed again and again. Recall that our subsets to be multimerged are each of size $CacheSize/2$, and therefore the heads of these subsets map to locations $CacheSize/2$ apart in the cache. The result is that the heads of the odd subsets all map to the same cache block, as do all the heads of the even subsets. Consider the behavior of the first cache block of the first sorted subset. The block will be faulted in to read its first key and will be ejected when the first key of the third subset was added to the queue. The block will then to be faulted in again to read its second key and will be ejected again if any of the other odd subsets need their second keys. Since the multimerge does not consume the subsets at exactly the same rate, the indices into the subsets will likely disperse in the cache, and the thrashing will subside. In its initial stages, however, this type of multimerge will have extremely poor cache performance. This problem can be fixed by adding an entire cache block's worth of keys to the queue at a time rather than just one. This introduces redundant computation, as the priority queue will have to re-sort the block's worth of keys, but

it eliminates the thrashing. The first block of the first subset will still be ejected by the first block of the third subset, but this is not important since all of its keys will be in the priority queue already, and it will not be needed again.

For a priority queue, my multimerge uses a 4-heap. My multimerge initially adds one block's worth of keys from each subset to the heap. The last key from each cache block is tagged with a special flag when it is added to the heap indicating that another block's worth of keys should be read from its subset. The multimerge repeatedly removes the minimum from the heap and places it in the destination array. If the removed heap element is tagged indicating that it is the last key from a block, another block's worth of keys is read from the subset that that element came from and is added to the heap. The multimerge does this repeatedly until the fully sorted set is complete.

I call the mergesort with a tiled first phase and a multimerge second phase *multi-mergesort*. This algorithm has two disadvantages when compared to the tiled mergesort. The first disadvantage is that it is no longer stable[2] since the keys are added to a heap and heaps are not stable. The second disadvantage is that it will have a higher instruction cost than the tiled mergesort, since it has been shown that sorting with a heap is less efficient from an instruction cost perspective than merging [Knuth 73]. This is compounded by the redundant computation that the multimerge introduces by adding blocks of keys to the heap that are already sorted.

The disadvantages of multi-mergesort, however, are offset by excellent cache behavior. The auxiliary heap will remain small and should only occasionally be ejected from the cache by sweeps through the source and auxiliary array. In the first phase, each cache block in both the source array and the auxiliary array will be faulted in once. Similarly in the second phase, each block in both the source array and auxiliary array will be faulted once, totaling only four cache misses per block's worth of keys. Thus on average $4(KeySize/BlockSize)$ cache misses should be incurred per key. For a 32 byte block size and 64 bit keys, this averages out to one cache miss per key.

---

[2] A sorting algorithm is *stable* if it preserves the relative order of keys with the same value [Knuth 73].

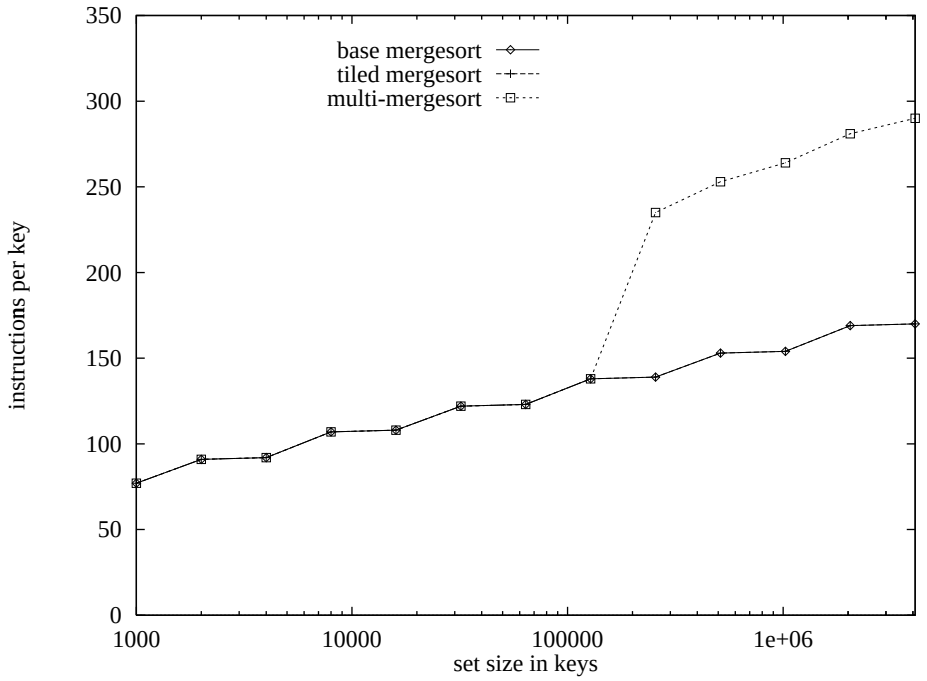

Figure 6.6: Instruction counts for mergesort.

6.2.3 Performance

As with heapsort, the performance of mergesort is measured by sorting sets of uniformly distributed 64 bit integers. Figure 6.6 shows a graph of the dynamic instruction count of the base mergesort, the tiled mergesort and the multi-mergesort. As expected, the base mergesort and the tiled mergesort execute almost the same number of instructions. The wobble in the instruction count curves in this graph is due to the final copy that may need to take place depending on whether the final merge wrote into the source array or the auxiliary array. When the set size is smaller than the cache, the multi-mergesort behaves exactly like the tiled mergesort. Beyond that size, the multimerge is performed and this graph shows the increase it causes in the instruction count. For 4,096,000 keys, the multimerge executes 70% more instructions than the other mergesorts.

Figure 6.7 shows the cache misses per key for the three mergesort algorithms for a 2 megabyte cache with a 32 byte block size. The most striking feature of this graph is

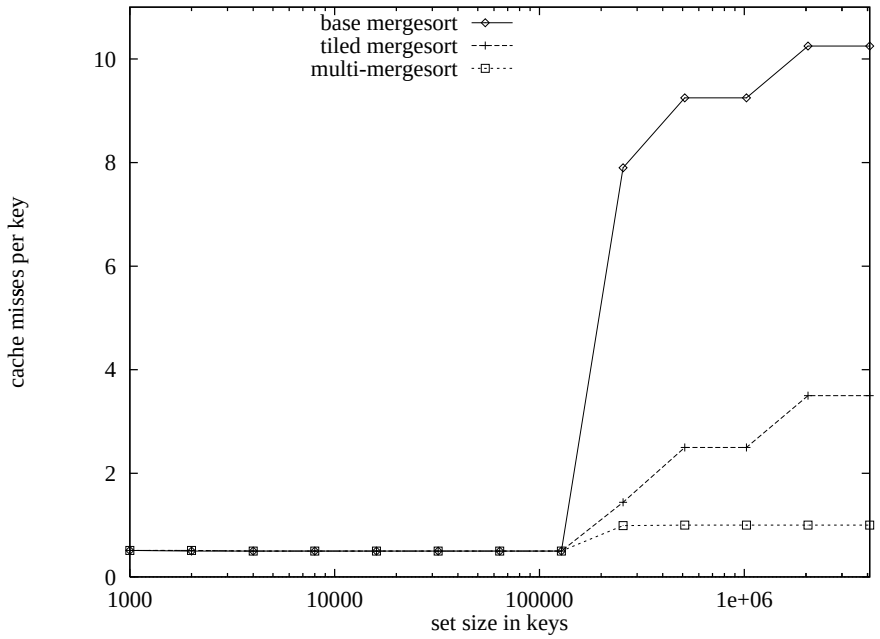

Figure 6.7: Cache performance of mergesort. Simulated cache size is 2 megabytes, block size is 32 bytes.

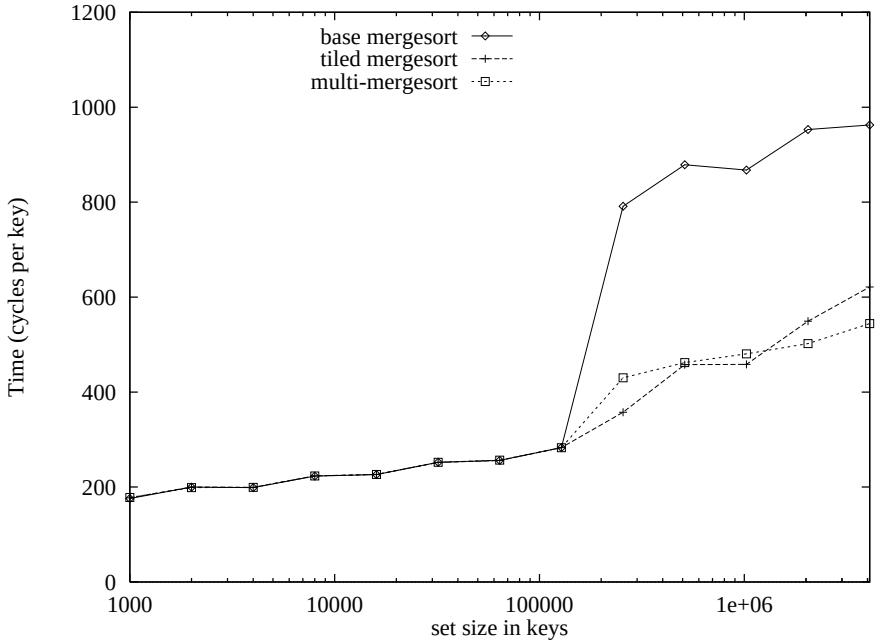

Figure 6.8: Execution time for mergesort on a DEC Alphastation 250.

the sudden leap in cache misses incurred by the base mergesort. The sharp increase in cache misses is due to the total elimination of temporal reuse that occurs when the source array and auxiliary array grow larger than the cache. Beyond that size, the misses per key increase with the log of the set size due to the need for more passes through the array. This graph shows the large impact that tiling the base mergesort has on cache misses. For 4,096,000 keys, the tiled mergesort incurs 66% fewer cache misses that the base mergesort. Once the large jump occurs in the miss curve for the base mergesort, the misses for these two algorithms increase at the same rate since they are using the same algorithm for the second phase. The wobble in these curves is again due to the possible need to copy the temporary array back into the source array at the end of the sort. The multi-mergesort is a clear success from a cache miss perspective, incurring no more than 1.002 cache misses per key, very close to our optimistic prediction of 1.

Figure 6.8 shows the execution time in cycles per key for the mergesorts on a DEC Alphastation 250. Up to the size of the second level cache, all of these algorithms perform the same. Beyond that size, the base mergesort performs the worst due to the large number of cache misses it incurs. The tiled mergesort executes up to 55% faster than the base mergesort, showing the significant impact the cache misses in the first phase have on execution time. When the multi-way merge is first performed, the multi-mergesort performs worse than the tiled mergesort due to the increase in instruction count. Due to lower cache misses, however, the multi-mergesort scales better and outperforms the tiled mergesort for the largest set sizes.

6.3 Quicksort

Quicksort is an in-place divide-and-conquer sorting algorithm considered by most to be the fastest comparison-based sorting algorithm when the set of keys fit in memory [Hoare 62]. In quicksort, a key from the set is chosen as the *pivot*, and all other keys in the set are compared to this pivot. A set of keys is partitioned around the pivot by dividing the set into those keys less than the pivot and those greater than the pivot. This is usually accomplished by walking through an array of keys from the outside in, swapping keys on the left that are greater than the pivot with keys on the right that are less than the pivot. At the end of the pass, the set of keys will be partitioned around the pivot and the pivot is guaranteed to be in its final position. The quicksort

algorithm then recurses on the region to the left of the pivot and the region to the right. The simple recursive quicksort can be expressed in less than twenty lines of code. I now examine explore efficient implementations of quicksort.

6.3.1 Base Algorithm

An excellent study of fast implementations of quicksort was conducted by Sedgewick, and I use the optimized quicksort he develops as my base algorithm [Sedgewick 78]. I now briefly describe the three main optimizations that are applied to the simple recursive quicksort to convert it to the implementation recommended by Sedgewick.

Sedgewick advocates using an iterative quicksort rather a recursive one due to the expense of performing procedure calls. By adding an auxiliary stack to keep track of the algorithm's state, the recursion can be turned into a loop which pushes and pops descriptions of the work to be performed. This conversion from a recursive to an iterative algorithm does not change the order in which the keys are accessed.

Sedgewick's second suggestion is that the pivot be chosen as the median of three random keys. This is a common optimization of quicksort and was first suggested by Singleton [Singleton 69]. The quality of a partition pass depends on how evenly sized the resulting subsets are. A bad choice of pivot can result in partitioning the set so that 99% of the keys are in one subset and only 1% are in the other. The closer the pivot is to the median of the list, the more balanced the two subproblems will be and the faster the algorithm will sort. Rather than pick a key as the pivot at random, Sedgewick's implementation selects the median of the first, middle and last keys in the set as the pivot. The effect that this optimization has on the expected instruction cost of quicksort has been well studied [Sedgewick 77].

Sedgewick's final optimization is to sort small subsets using insertion sort rather than quicksort, as we already did with mergesort. To minimize instruction costs, Sedgewick advocates not sorting any of the small subsets until the end. In his optimized quicksort, a subset to be sorted is ignored if its size is less than a threshold value. At the end of the quicksort, a sentinel is placed at the end of the array, and an insertion sort pass is made through the entire set to sort the small subsets. The sentinel allows a bounds check to be eliminated in the insertion sort resulting in an efficient algorithm for sorting the small subsets. This bounds check could not have been eliminated had the small subsets been sorted individually when they were

popped off the work stack.

6.3.2 Memory Optimizations

In practice, quicksort generally exhibits excellent cache performance. Since the algorithm makes sequential passes through the source array, all keys in a block are always used and spatial locality is excellent. Quicksort's divide-and-conquer structure also gives it excellent temporal locality. If a subset to be sorted is small enough to fit in the cache, quicksort will incur at most one cache miss per block before the subset is fully sorted. Despite this, an examination of the memory behavior of the base quicksort reveals two ways in which its cache performance can be improved.

The first memory optimization is to remove Sedgewick's elegant insertion sort at the end and instead sort each small subset when it is first encountered. While saving them all until the end makes sense from an instruction cost perspective, it is exactly the wrong thing to do from a cache performance perspective. When quicksort pops a small subset to sort off of its work stack, it is highly likely that all of the keys in this subset will be in the cache since they were just partitioned. Sorting small subsets with an unoptimized insertion sort at the time that they are popped off of the work stack will avoid any cache misses. In the base quicksort on the other hand, the insertion sort pass at the end needs to load every key into the cache. If the set size is larger than the cache, this will incur a substantial number of cache misses. I call the base quicksort with this optimization applied the *memory-tuned quicksort.*

The second memory optimization is to have quicksort perform a multi-way partition similar to the multi-way merge used in multi-mergesort. Although quicksort incurs only one cache miss per block when the set is cache-sized or smaller, larger sets incur a substantial number of misses. The initial passes made by quicksort suffer from the same problem as the second phase of the tiled mergesort algorithm. To fix this inefficiency, a single multipartition pass is used to divide the full set into a number of subsets which are likely to be cache sized or smaller.

Multipartitioning is used in parallel sorting algorithms to divide a set into subsets for the multiple processors [Blelloch et al. 91, Hui & Sevcik 94]. The performance of these parallel sorts depends on the work being balanced evenly between processors, and there are complex pivot selection algorithms to divide up the keys as evenly as possible. I use a simpler approach and choose the number of pivots so that the number

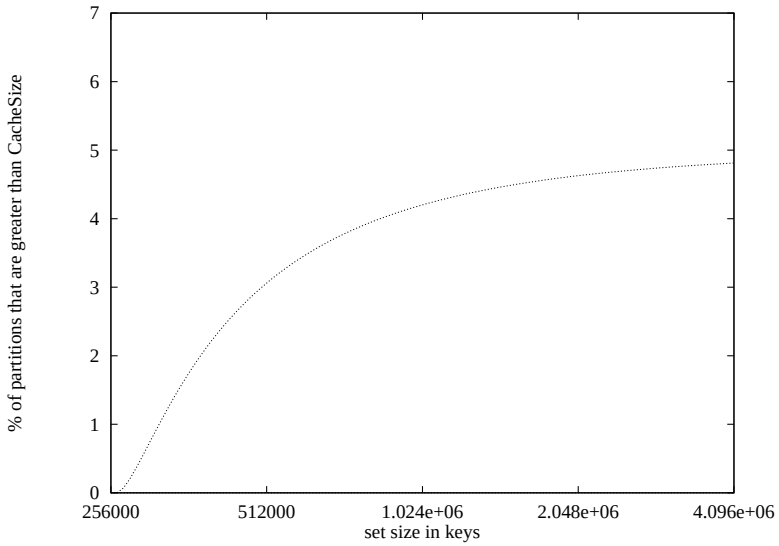

Figure 6.9: The chance that a partitioned subset is greater than the cache.

of subsets larger than cache is small on average. I partition the set into k subsets where k is chosen so that the average subset size is $CacheSize/3$, and analysis shows that the chance that a given subset is larger than the cache is less than 5%. Feller shows that if y partitions are placed randomly in a range of length 1, the chance of a resulting subrange being of size x or greater is exactly $(1-x)^y$ [Feller 71, Vol. 2, Pg. 22]. Let N be the total number of keys and C be the cache size in keys. To divide the set so that the average subset is of size $C/3$, it needs to be partitioned into $3N/C$ pieces, requiring $(3N/C)-1$ pivots. Feller's equation indicates that after the multipartition, the chance that a subset is larger than C is $(1-C/N)^{(3N/C)-1}$. Figure 6.9 shows this expression for $C = 262,144$ while N is varied from 256,000 to 4,096,000. This graph shows that with this simple heuristic, the chance that a resulting subset is larger than the cache is small. In the limit as N grows large, the percentage of subsets that are larger than the cache is e^{-3}, less than 5%.

The multipartition requires a number of auxiliary data structures. Unlike a binary partition, a k-way partition cannot easily be performed in place, and instead, a temporary list is allocated for each of the k subsets. A difficulty is that the size of the resulting subsets is not known ahead of time. In a pathological case, one list

may contain all but $k-1$ of the keys. To alleviate this problem, the temporary lists are implemented as linked lists of blocks of keys. The lists start with a single block of keys, and new blocks are allocated and linked together as needed. The number of keys per block was varied between 100 and 5,000 and had very little impact on performance. In addition to these auxiliary lists, an array of size $k-1$ is needed to store the pivots.

The multipartition quicksort executed the following algorithm. It first chooses $k-1$ pivots, sorts them, and stores them in the pivot array. Each key in the source array is then moved to the temporary list holding the appropriate subset, and the subset to which a key belongs is determined by binary searching through the pivot array. After partitioning the entire source array it copies the keys from each of the temporary lists back into the source array and sorts each subset using the base quicksort. Since we expect the subsets to fit in the cache, it makes more sense to sort them with the base quicksort than with the memory-tuned quicksort, due to its lower instruction cost.

In practice, the memory-tuned quicksort does not incur a significant number of cache misses until the set size is more than twice the size of the cache. For this reason, the multipartition quicksort only performs the multipartition if the set is more than twice the size of the cache. I refer to the quicksort algorithm with the multipartition as *multi-quicksort*.

As with our multi-mergesort, multi-quicksort is less efficient from an instruction cost perspective than the base quicksort. Also like the multi-mergesort, we expect multi-quicksort to exhibit good cache behavior. Each block of keys will be faulted into the cache once when read from the source array and once when placed onto a temporary list. After the multipartition is finished, the keys will again be faulted when read from the temporary lists and again when writing back into the source array. If all of the k subsets are cache-sized or less, cache misses will not occur during the sorting of the subsets. This yields an optimistic total of four misses per source block, the same as the multi-mergesort. In practice, some of the subsets will be larger than the size of the cache, but since this is uncommon it should not significantly affect the number of misses per key.

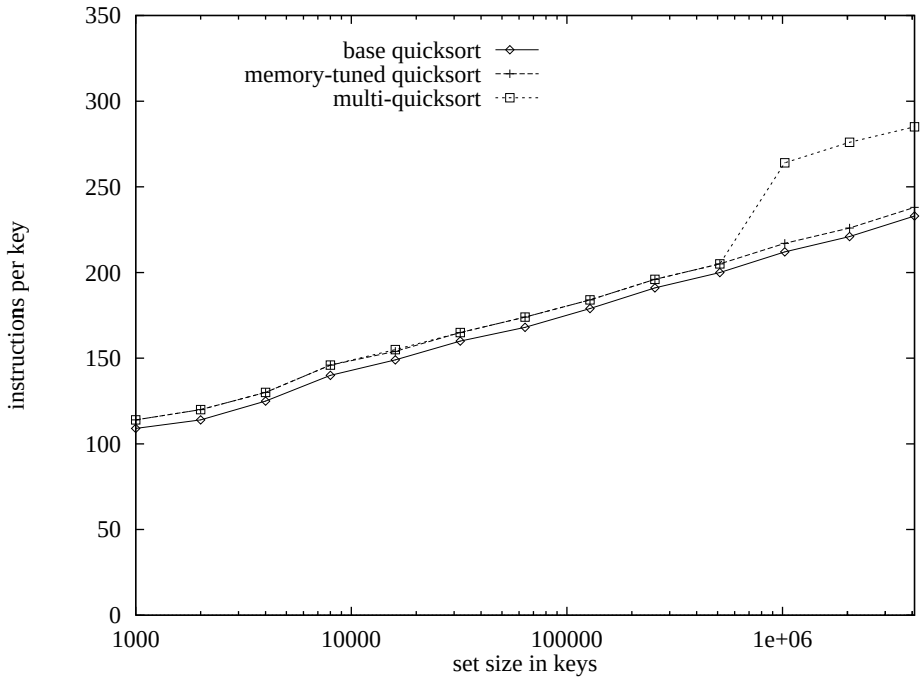

Figure 6.10: Instruction counts for quicksort.

6.3.3 Performance

Figure 6.10 shows the number of instructions executed per key for each of the three quicksort algorithms sorting 64 bit uniformly distributed integers. The base quicksort executes the fewest instructions with the memory-tuned quicksort executing a constant number of additional instructions per key. This difference is due to the inefficiency of sorting the small subsets individually rather than at the end as suggested by Sedgewick. When the set size is greater than twice the cache size, the multi-quicksort performs the multipartition, and this graph shows that the multi-quicksort executes up to 20% more instructions than the memory-tuned quicksort.

Figure 6.11 shows the cache performance of the three quicksort algorithms. This graph shows that all of the quicksort algorithms incur very few cache misses. The base quicksort incurs fewer than two misses per key for 4,096,000 keys, lower than all of the other algorithms up to this point with the exception of the multi-mergesort. Memory-tuned quicksort eliminates a constant 0.25 misses per key for large set sizes by sorting

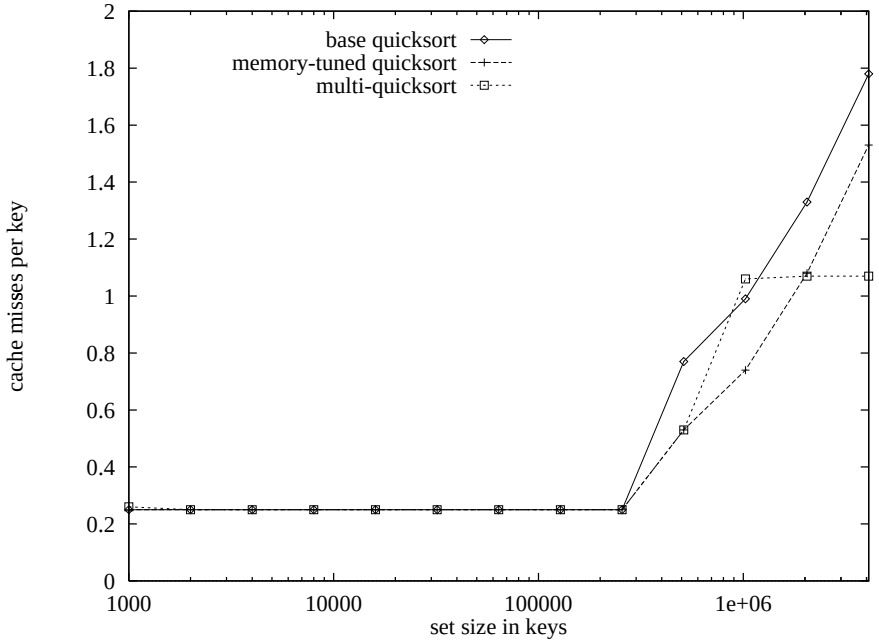

Figure 6.11: Cache performance of quicksort. Simulated cache size is 2 megabytes, block size is 32 bytes.

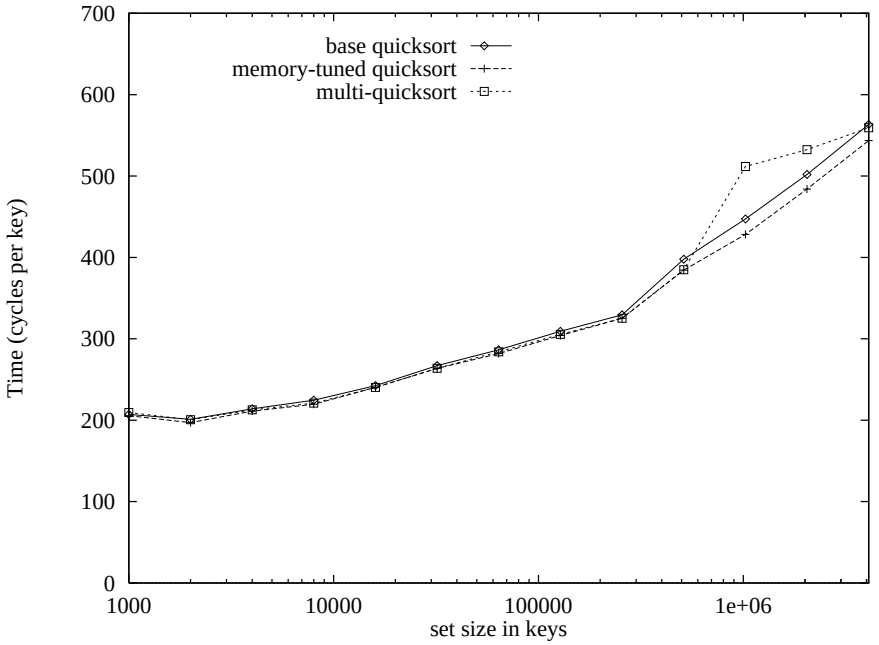

Figure 6.12: Execution time for quicksort on a DEC Alphastation 250.

small subsets early. This graph also shows that the multi-way partition produces a flat cache miss curve much the same as the curve for the multi-mergesort. The maximum number of misses incurred per key for the multi-quicksort is 1.07, validating the conjecture that it is uncommon for the multipartition to produce subsets larger than the size of the cache.

Figure 6.12 shows the execution times of the three quicksort algorithms. All three of these algorithms perform similarly on the DEC Alphastation 250. This graph shows that sorting small subsets early is a benefit, and the reduction in cache misses outweighs the increase in instruction cost. The multipartition initially hurts the performance of the multi-quicksort due to the increase in instruction cost, but the low number of cache misses makes it more competitive as the set size is increased. 4,096,000 keys was chosen as the maximum set size due to the physical memory limits on the machine I used. This graph suggests that if more memory were available and larger sets were sorted, the multi-quicksort would outperform both the base quicksort and the memory-tuned quicksort.

6.4 Comparison

To compare the performance of the eight sorting algorithms from this chapter, the performance graphs for the heapsorts, mergesorts and quicksorts are combined together. The combined graphs for instruction count, cache misses and cycles executed per key are shown in Figures 6.13-6.15.

The instruction count graph shows that the heapsorts execute the most instructions, while the mergesorts execute the least. It might be surprising that the mergesorts execute fewer instructions than the quicksorts. Sedgewick's analysis supports this result, indicating that quicksort's inner loop executes roughly the same number of instructions as the optimized mergesort's and is executed 38% more often [Sedgewick 88]. Mergesort also has a very regular structure with no data-dependent control flow which makes it more amenable to traditional optimization that quicksort. In the mergesorts, sorted lists of size four were initially created with a very efficient inline sort. This was not possible with quicksort, as the small subsets varied in size, and a non-inline sort was needed. The inner loop of mergesort was also very amenable to unrolling, which eliminates the majority of the loop overhead. Since the number of swaps performed by quicksort each partition pass is not known ahead of time, the

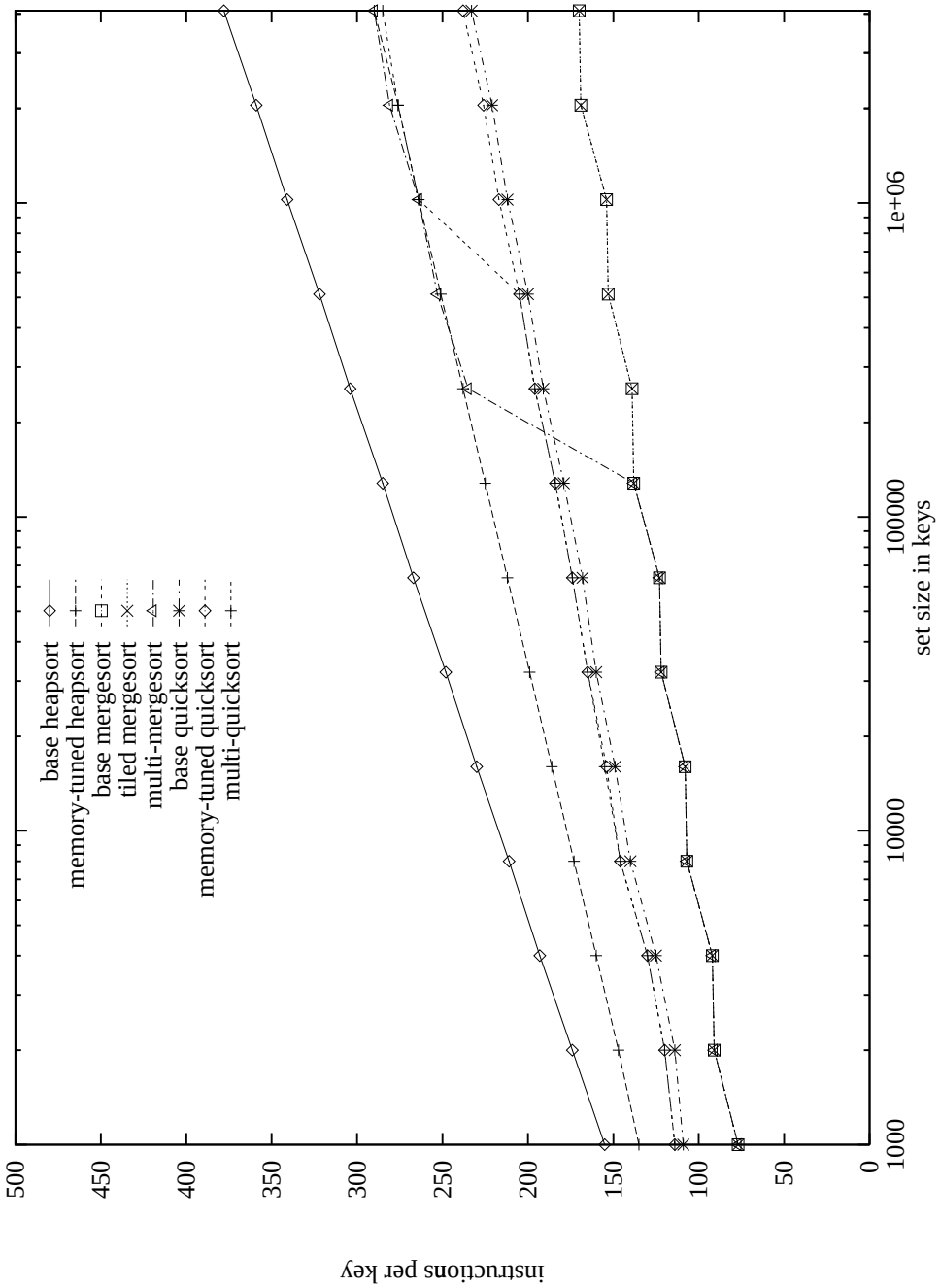

Figure 6.13: Instruction counts for eight sorting algorithms.

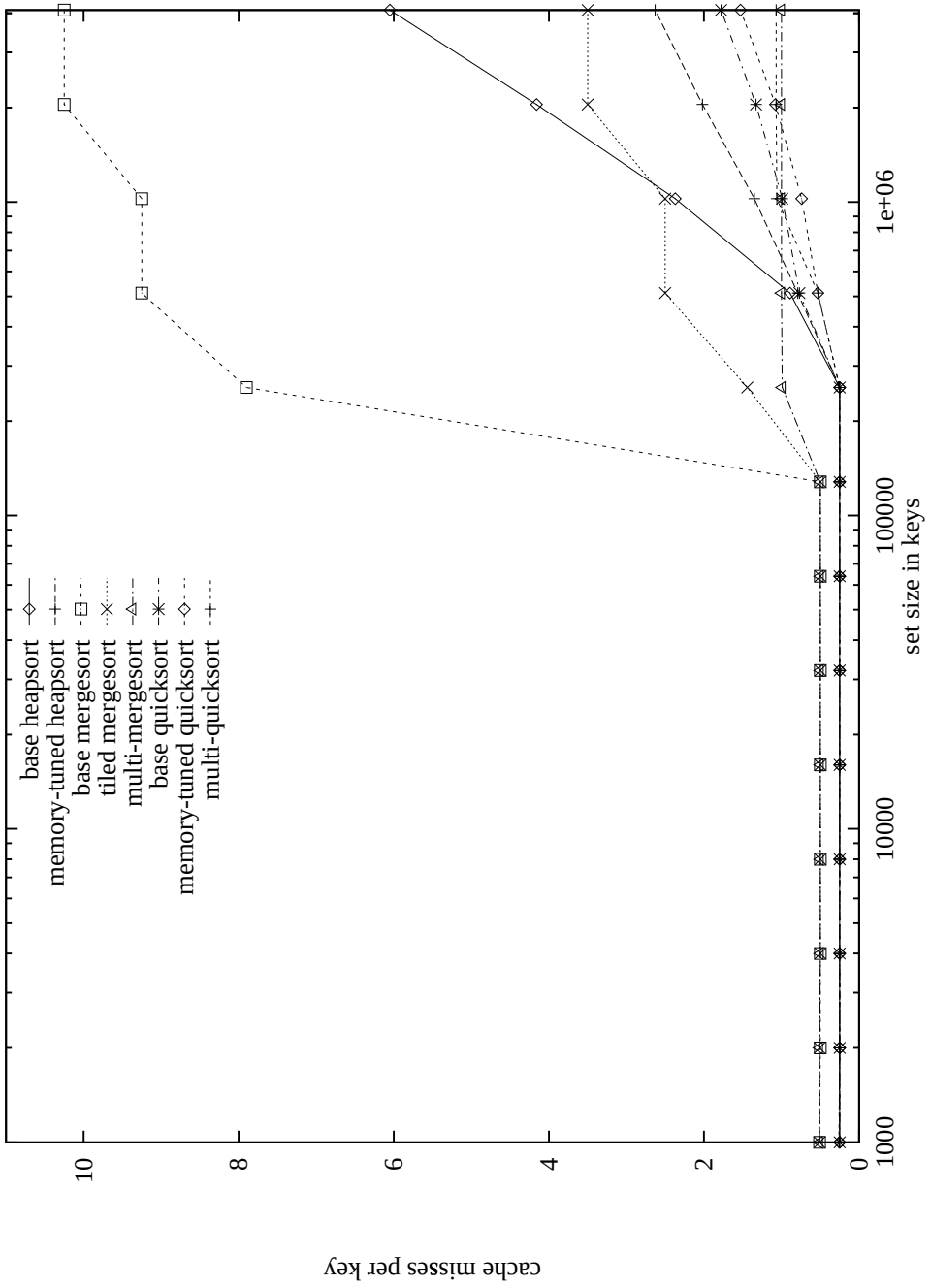

Figure 6.14: Cache performance of eight sorting algorithms. Simulated cache size is 2 megabytes, block size is 32 bytes.

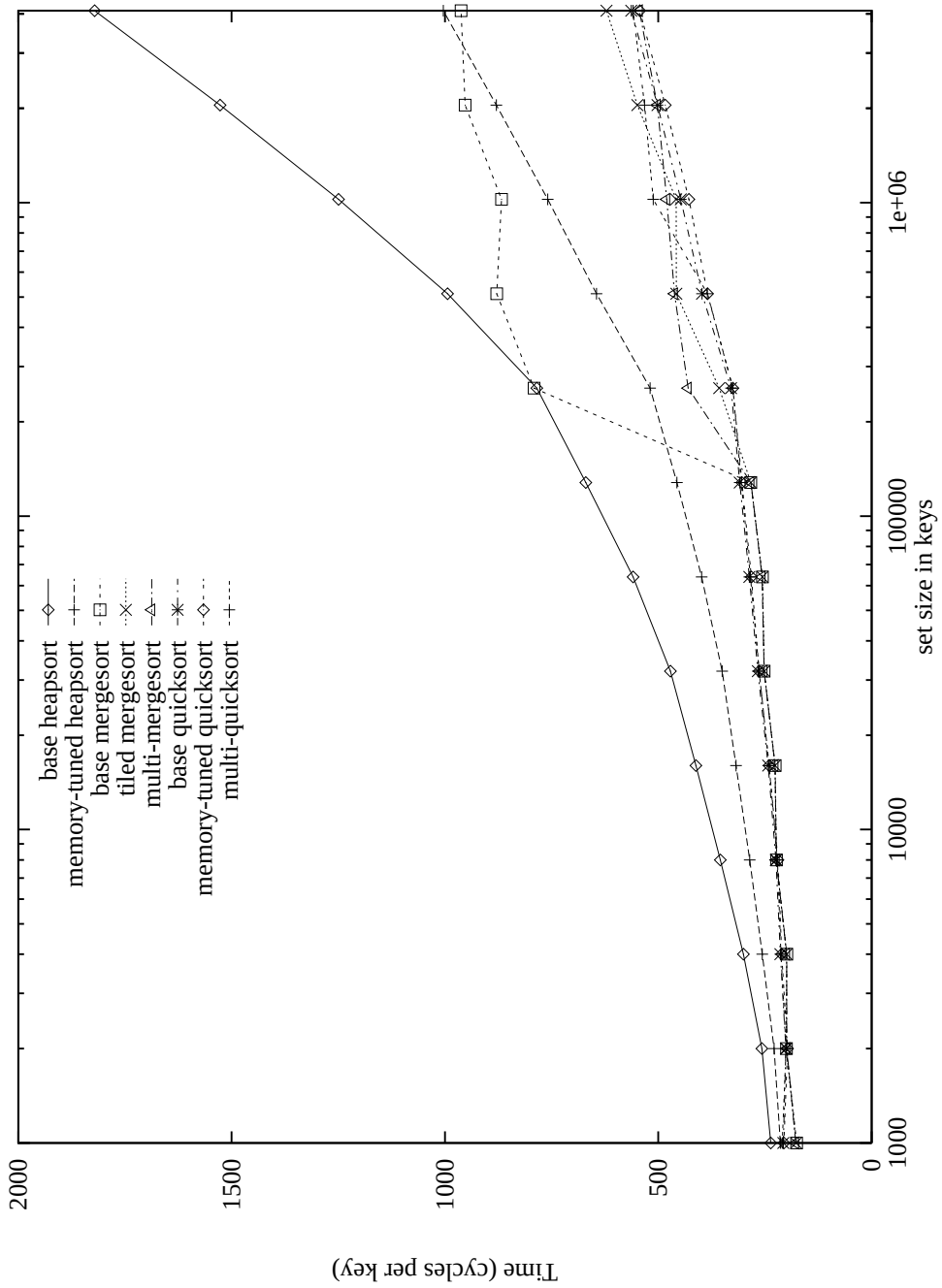

Figure 6.15: Execution time for eight sorting algorithms on a DEC Alphastation 250.

loop overhead of the partition pass can not easily be amortized using loop unrolling.

The cache miss graph shows that the memory optimizations significantly reduce the cache misses incurred for large data sets. For 4,096,000 keys, cache misses are reduced by up to 57% for heapsort, 90% for mergesort and 40% for quicksort. This graph also shows the difference between the in-place and non-in-place sorting algorithms. The in-place algorithms incur only 0.25 compulsory misses per key and do not start incurring non-compulsory cache misses until the set size is greater than the cache size ($n > 262,144$). The non-in-place sorts, on the other hand, incur 0.5 compulsory misses per key since they require an auxiliary array the same size as the source array. This also causes them to start incurring non-compulsory misses when the set size is greater than half the cache size ($n > 131,072$). It is interesting to see that neither of the two algorithms with the best cache performance sort in-place. Both the multi-mergesort and the multi-quicksort use auxiliary structures at least as large as the source array. My belief at the beginning of this study was that an in-place algorithm would have the best cache performance. This study shows that although these two algorithms use twice as much memory as the in-place algorithms, they use the cache more efficiently and as a result incur fewer cache misses.

The graph of execution time shows that all of the base algorithms are outperformed by memory-tuned algorithms for large set sizes. The 4-heap sorts up to 82% faster than the base heapsort, the multi-mergesort sorts up to 75% faster than the base mergesort and the memory-tuned quicksort sorts up to 4% faster than the base quicksort. While implicit heaps may be an excellent priority queue, this graph shows that they cannot compete with the other algorithms for sorting. The 4-heap has a lower instruction count and better spatial locality than the traditional heap, but it still executes more instructions and has worse temporal locality than the memory-tuned mergesorts and quicksorts. The two memory tuned mergesorts and all three of the quicksorts perform well, and their execution time curves are tightly grouped on this graph. If stable sorting is not important, the base quicksort and the memory-tuned quicksort are the best choices since they are in-place and can sort larger sets of keys without the danger of paging. The tiled mergesort is only 10% slower than the base quicksort, making it a good choice if a stable sorting algorithm is needed. Neither the multi-mergesort or the multi-quicksort are in-place or stable. Nevertheless, these two algorithms offer something that none of the others do. They both incur

very few cache misses which renders their overall performance far less sensitive to cache miss penalties than the others. As a result, these algorithms can be expected to outperform the others as relative cache miss penalties continue to increase.

6.5 Summary

This chapter presents a study of the cache performance of heapsort, mergesort and quicksort. For all three of these, a base algorithm is tuned using traditional optimization techniques. These base algorithms are then examined for inefficiencies in their memory behavior and appropriate memory optimizations are applied.

The performance of the algorithms presented in this chapter argues strongly for both design and analysis techniques that take caching into account. For heapsort, mergesort and quicksort, the execution time of the base algorithm was improved by applying simple memory optimizations. Of the eight sorting algorithms presented, the two slowest for 4,096,000 keys were base variants while five of the six fastest were memory-tuned variants. This shows that the optimization of both instruction cost and memory behavior offers better overall performance than optimizing instruction cost alone. It is important to recognize that these gains would not have been predicted by unit cost analyses. The multi-mergesort, for instance, executes 70% more instructions that the base mergesort, clearly inferior from a unit-cost perspective, yet sorts up to 75% faster on an Alphastation 250. Only by utilizing analysis techniques that account for the effects of caching can the performance of these algorithms be fully understood and the best possible performance be realized.

Chapter 7

Lock-Free Synchronization

Throughout this thesis, I have examined the interaction between caches and algorithms. Up to this point, I have only examined this interaction in the context of sequential algorithms. I now investigate whether the same analysis and optimization techniques can be applied to a class of parallel algorithms for shared-memory multiprocessors.

Like sequential machines, shared-memory multiprocessors have fast local caches, and missing in these caches results in long delays as the data is read over the interconnection network. Despite this, most analyses of parallel algorithms are performed in models in which all memory accesses have unit cost. There are number of parallel machine models that differentiate local memory references from remote references [Snyder 86, Culler et al. 93]. Unfortunately, the majority of analyses of parallel algorithms are performed in the unit-cost PRAM model [Fortune & Wyllie 78].

There are important differences between parallel and sequential machines that need to be accounted for if memory system performance is to be understood. On parallel machines, multiple processors can access the interconnection network at the same time, creating *contention* on the network. The time to service a cache miss on a parallel machine is partially dependent on this contention. Unlike sequential machines in which cache miss penalties are approximately constant, miss penalties on parallel machines depend on the behavior of the other processors. Another important difference is that parallel machines offer synchronization instructions that are used to coordinate the sharing of data between processors. These synchronization instructions often invoke data coherency mechanisms which make use of the interconnection network, further impacting the memory system performance of the other processors

in the system.

To investigate how the techniques from the previous chapters can be applied to parallel algorithms, I examine the memory system performance of lock-free synchronization protocols. These are an important and relatively new class of concurrent objects that allow parallel algorithms to synchronize without the use of locks. Unfortunately, existing lock-free synchronization protocols are slower than locks in the common case, and this has prevented their adoption in practice.

In this chapter, I present a model for investigating the memory system performance of lock-free synchronization protocols. This performance model reflects three important architectural characteristics: first, that cache misses which must access remote memory are more expensive than cache hits, second, that synchronization instructions can be even more expensive than cache misses, and third, that optimistic synchronization policies result in unsuccessful thread updates which consume communication bandwidth and slow the progress of the other threads. To validate the importance of these characteristics, the model's predictions are compared with the results of parallel machine simulations. The predictions of relative performance are fairly accurate, suggesting that these characteristics are an important factor in the overall performance of lock-free synchronization protocols.

The analysis in this chapter indicates that no existing protocol provides insensitivity to common delays while still offering performance equivalent to locks. Accordingly, I introduce a protocol based on a combination of existing lock-free techniques that provides low latency and insensitivity to preemption delays.

7.1 Background

Threads running on shared-memory multiprocessors coordinate with each other via shared data structures called *concurrent objects*. In order to prevent the corruption of these concurrent objects, threads need a mechanism for synchronizing their access to them. Typically, this synchronization is provided with locks protecting critical sections, ensuring that at most one thread can access an object at a time. Concurrent threads are difficult to reason about, and critical sections greatly simplify the building of a correct shared object. While much work has been done to improve the performance of locks [Anderson 90, Graunke & Thakkar 90, Mellor-Crummey & Scott 91], critical sections are not ideally suited for asynchronous systems. The delay or failure

of threads holding locks can severely degrade performance and cause problems such as *convoying*, in which parallelism is not exploited as threads move together from lock to lock, *priority inversion*, in which a high priority thread cannot make progress because it needs a lock being held by a low priority thread that has been preempted, and *deadlock*, which occurs when all active threads in the system are waiting on locks [Zahorjan et al. 88]. In the extreme case, the delay of a single thread holding a single lock can prevent all other threads from making progress. Sources of these delays include cache misses, remote memory accesses, page faults and scheduling preemptions.

Recently, researchers have proposed concurrent objects that synchronize without the use of locks. As these objects are built without locks, they are free from the aforementioned problems. In addition, lock-free objects can offer progress guarantees. A lock-free object is *non-blocking* if it guarantees that some thread completes an operation in a finite number of steps [Herlihy 90]. A lock-free object is *wait-free* if it guarantees that each thread completes an operation in a finite number of steps [Herlihy 90]. Intuitively, the non-blocking property says that adversarial scheduling cannot prevent all of the threads from making progress. The wait-free property is stronger and says that starvation of any thread is not possible regardless of the potential interleavings of operations. Johnson has modeled the performance advantages that lock-free synchronization offers over traditional locking when different types of delays occur [Johnson 95].

There has been a considerable amount of research supporting the practical use of lock-free synchronization. Herlihy and Moss have proposed architectural changes that enable efficient lock-free computing [Herlihy & Moss 93]. Bershad has described a technique that uses operating system support to simulate important synchronization instructions when they are unavailable in the architecture [Bershad 93]. Numerous lock-free implementations of specific data-structures have been proposed including Wing and Gong's object library [Wing & Gong 90], Massalin's lock free objects [Massalin & Pu 89], Valois's linked structures [Valois 95], and Anderson and Woll's union-find object [Anderson & Woll 91]. In addition, a number of software protocols have been developed that generate lock-free concurrent objects given a traditional sequential implementation of the object. Herlihy's small object protocol and wait-free protocol [Herlihy 91], Anderson and Moir's large object protocol [Anderson & Moir 95], Alemany and Felten's protocols [Alemany & Felten 92], and

Barnes's caching method [Barnes 93] fall in this category. Since these protocols build concurrent objects with no knowledge of the object's semantics, I call them *black-box* synchronization protocols. These black-box protocols are of practical interest since they can be applied to any sequential object and could be automatically applied by a compiler. Unfortunately, these protocols do not perform as well as traditional lock-based synchronization in practice. This chapter examines the performance characteristics of these black-box synchronization protocols.

7.2 Performance Issues

A common criticism of lock-free synchronization techniques is that while they offer real-time and fault-tolerant benefits over locks, these benefits are not realized when there are no thread delays and that in actual use, they perform poorly. In general, lock-free protocols have higher latency and generate more memory contention than locks. In this section, I describe three important architectural characteristics that affect the performance of lock-free synchronization protocols.

1. *Cache misses, which must access remote memory, are more expensive than cache hits*

   With few exceptions, modern parallel machines have fast local caches, and accessing these caches is one to two orders of magnitude faster than accessing main memory. In addition, depending on the coherency protocol, cache hits do not put a load on the communication medium used to support shared memory, unlike cache misses. Given that caching can have such a large impact on performance, we would not expect a PRAM-style analysis in which all memory accesses have unit cost to accurately predict performance.

   The idea of treating remote and local references differently is not a new one. Many models, including Snyder's CTA and the LogP model include this distinction [Snyder 86, Culler et al. 93]. The notion of reducing the number of non-local memory references during synchronization has also been previously investigated. Anderson and Mellor-Crummey and Scott consider local versus remote accesses when explaining the contention caused by *Test*-and-*Test&Set* locks and in the design of their queue-based spin locks [Anderson 90, Mellor-Crummey & Scott 91].

2. Synchronization instructions are even more expensive that cache misses

Lock-free synchronization protocols make use of synchronization instructions such as *Test&Set*, *Compare&Swap* and the combination of *Load Linked* and *Store Conditional*. On current machines these synchronization instruction incur a cycle cost much higher than that of a normal memory reference. For example, on an Alpha 3000/400 with a 130 MHz Alpha 21064 CPU [Dig 92], I observed a cost of 140 cycles for the pair of *Load Linked* and *Store Conditional* instructions, 3.5 times more expensive than a normal uncached read. This property is not exclusive to this particular architecture, and synchronization instructions for many modern processors incur similar costs [Bershad 93, Bershad et al. 92]. I do not distinguish synchronization from non-synchronization instructions because of an inherent difference in complexity, but rather because of the implementation differences that occur in practice.

This distinction is also important when the necessary synchronization instructions are unavailable on an architecture and must be simulated by the operating system [Bershad 93]. In these situations, the code executed to simulate the desired instruction can take much longer than a single memory operation and needs to be taken into account. Lastly, distinguishing synchronization instructions from non-synchronization instructions is important on distributed shared memory systems such as Munin [Carter et al. 91] or Midway [Bershad et al. 93] or shared memory multiprocessors such as Dash [Lenoski et al. 92] that support a consistency model looser than sequential consistency. In these systems, synchronization instructions invoke the coherency mechanism which in turn results in costly communication.

Again, I find that analyzing synchronization algorithms in a PRAM-style model that assigns all instructions unit cost can introduce unnecessary inaccuracy.

3. Optimistic synchronization policies result in unsuccessful thread updates that consume communication bandwidth and slow the progress of the other threads in the system

In order to be non-blocking, a number of the lock-free protocols behave optimistically. That is, all threads proceed as if they will succeed. Once threads realize that their update has failed, either they begin another attempt or they cooperate in order to help the successful thread [Barnes 93, Herlihy 91]. This optimistic policy results in the waste of machine resources when there is contention for a shared object. Processor

cycles are wasted that could possibly be used by another thread. More importantly, communication bandwidth is wasted which in turn slows down the cache misses of the successful thread. As more unsuccessful threads contend for the communication bandwidth, the progress of the system as a whole can be crippled. This is what Alemany and Felten refer to as "useless parallelism" [Alemany & Felten 92]. Herlihy attempts to alleviate this problem by using exponential backoff to reduce the number of unsuccessful updates [Herlihy 90].

The performance degradation caused by an optimistic policy is important and should be quantified in a good performance model.

I now develop a simple analytical model for predicting black-box protocol performance that takes these characteristics into account. Comparing the predictions of this model with simulated execution results allows me to validate the model which in turn validates the importance of these characteristics.

7.3 Performance Model

I now present a simple performance model that reflects the cache miss costs, synchronization costs and wasted work costs discussed in the previous section. It is intended that this model be simple enough to evaluate algorithms quickly, yet still provide good insight into practical performance. This model can be used to explain the performance of existing protocols, determine how changes in architecture will affect protocol performance, and serve as a guide for designers of new lock-free protocol.

7.3.1 Model Variables

My performance model measures the amount of work done by a particular protocol in order to complete a single update to a shared object. The model assumes that the data caches start out cold and that all instruction fetches hit in the cache. I divide instructions into three categories. The first category contains local instructions such as adds, compares and memory accesses that hit in the cache. The second group contains memory accesses that miss in the cache and must access remote memory. The third group contains synchronization instructions such as *Compare&Swap* and *Store Conditional*.

In the model, local instructions have cost 0 in order both to keep the model simple

and to reflect their low cost. Instructions in the second group are given a normalized cost of 1. Lastly, in order to reflect their higher cost, synchronization instructions are given cost C.

In the model, N denotes the number of threads in the system. S represents the size of the sequential object's state in cache blocks. W denotes the number of cache blocks that are written in the course of an operation. R denotes the number of cache blocks that are read that are not also written. Reads and writes are differentiated in this model because some back-box protocols perform more work on writes than on reads. Since I only consider non-overlapping reads and writes, $R + W$ is less than or equal to S and could be significantly smaller. Enqueuing to a 100 word stack, for instance, might have $S = 13$, $R = 1$ and $W = 2$.

7.3.2 Workloads

I consider the performance of protocols in the presence of three different adversaries, each of which reflects a different workload. The first case to consider is when there is no contention and a single thread applies an operation to the concurrent object. Torrellas *et al.* found that on a 4 CPU multiprocessor running System V, threads accessing the six most frequently accessed operating system data structures found them unlocked from 85 to 99 percent of the time [Torrellas et al. 92]. While this says nothing about user code or large multiprocessors, it does suggest that well written systems are designed to minimize the contention on shared objects. In order to measure the latency of a protocol in this case, I introduce a weak adversary called *best* that allows a single thread to execute its operation to completion without blocking.

While we do not expect high contention to be the common case, it is still important to know how a protocol's performance degrades when an object is accesses by multiple threads concurrently. In order to model high contention, I include a *bad* scenario in which all N threads try to apply an operation concurrently and the adversarial scheduler can arbitrarily interleave their instructions.

A number of lock-free protocols rely on the operating system to execute code on a thread's behalf when it blocks, either on I/O or due to a scheduler preemption. For these protocols, there is an important distinction between the scheduler blocking a thread and the scheduler simply not allowing a thread to run. In order to measure the effect that an extremely powerful adversary can have on these protocols, I also include

Table 7.1: Total amount of work done to complete a single operation.

Method	**Best Case**	**Bad Case**	**Worst Case**
Spin-lock	$R+W+C+1$	$R+W+N(C+1)$	∞
Herlihy's small object	$S+C+3$	$N(S+C+3)$	$N(S+C+3)$
Herlihy's wait-free	$S+2N+C+3$	$N(S+2N+C+3)$	$N(S+2N+C+3)$
Alemany and Felten's solo protocol	$S+3C+4$	$N(C+1)+S$ $+2C+3$	$N(S+2C+3)+$ $\frac{N(N+1)}{2}(C+1)$
A and F's solo w/ log	$R+2W+C+1$	$R+2W+N(C+1)$	∞
Barnes's Caching Method	$R(4C+3)+$ $W(6C+4)-C-1$	$N(R(4C+3)+$ $W(6C+4)-C-1)$	$N(R(4C+3)+$ $W(6C+4)-C-1)$
Solo-cooperative	$R+W+C+1$	$R+W+N(C+1)$	∞

a *worst* scenario, in which all N threads are active and the adversarial scheduler can arbitrarily interleave their instructions and cause them to block. Once a thread is blocked, the worst case adversary has no obligation to wake the thread up.

7.4 Applying the Performance Model

I now evaluate seven synchronization protocols in the model: five existing lock-free protocols, a new lock-free protocol of my own design, and for comparison a spin-lock. The existing-lock free protocol I evaluate are Herlihy's small object protocol and wait-free protocol, Alemany and Felten's solo protocol and solo protocol with logging, and Barnes's caching method. A summary of the evaluation of the protocols in the model is shown in Table 7.1.

7.4.1 Spin-lock

Spin-locks can be used to implement critical sections, a straightforward way to synchronize accesses to concurrent objects. To update an object using a spin-lock, a thread first reads the lock's value to see it is free. If it is not free, the thread spins on the lock value, repeatedly reading it until it is free. Once the thread notices that the lock is free, it attempts to acquire the lock using a synchronization instruction

such as *Compare&Swap*. If the *Compare&Swap* succeeds, the thread owns the lock and can apply its operation to the object and release the lock. If the *Compare&Swap* fails, the thread returns to spinning on the lock.

In the best case scenario, the single thread reads the lock value at cost 1 and acquires the lock at cost C. The thread applies its operation at cost $R + W$, and since the lock value is cached, no cost in incurred by the thread to release the lock. This yields a total cost of $R + W + C + 1$ for the best case. In the bad case, the scheduler can cause the lock to be a point of contention, and it can force all N of the threads to read the lock value (cost N) and attempt to acquire the lock (cost NC). It then allows the successful thread to apply its operation to the object and release the lock (cost $R + W$), for a total bad case cost of $R + W + N(C + 1)$. In the worst case scenario the scheduler can prevent the spin-lock from ever completing an operation. To achieve this, the worst case scheduler simply needs to allow a thread to acquire the lock and then block the thread. If the scheduler never wakes this thread up, no other threads can make progress, and the protocol is deadlocked. While this can easily be caused by an adversarial scheduler, it can also happen in a multiprocessor due to process or processor failure.

7.4.2 Herlihy's Small Object Protocol

Herlihy's small object protocol is the earliest block-box technique for synchronizing accesses to concurrent objects without the use of locks [Herlihy 90, Herlihy 91]. In the small object protocol, all references to the concurrent object are made through a single shared pointer that points to the object's state. Updates are not directly applied to this version of the object. Instead, the object is copied, the operation is applied to this copy, and if no other threads have changed the shared pointer since the copy was taken, the shared pointer is changed to point at this modified copy. In the event that the pointer has changed since the copy was performed, the update is considered to be a failure, a new copy is made, and the process is attempted again. Although individual threads may never make progress due to repeated failures, the system as a whole is guaranteed to make progress since a failed update is always caused by another thread's successful update.

The small object protocol can be implemented with any universal synchronization primitive, and I consider an implementation using the combination of *load-linked* and

store-conditional as recommended by Herlihy. To update the shared object, threads *load-linked* the shared pointer and make a local copy of the object's state. A parity word at the beginning and end of the local copy need to be checked to validate the copy. The thread then applies its operation to the local copy of the object. Finally, the thread attempts to install the local copy with a *store-conditional* to the shared pointer. If the *store-conditional* fails, the thread begins again otherwise it returns having successfully updated the shared object. In the best case scenario, the *load-linked*[1] incurs cost 1, the copy incurs cost S, the validate incurs cost 2, and the *store-conditional* incurs cost C, for a total cost of $S + C + 3$. In both the bad and worst case, the scheduler can force all N threads to *load-linked* the pointer, copy and validate the object, apply their operation and attempt the *store-conditional* for a total cost of $N(S + C + 3)$ per successful operation.

7.4.3 Herlihy's Wait-Free Protocol

Herlihy's wait-free protocol offers a stronger progress guarantee than the small object protocol by incorporating a technique called *operation combining*. Operation combining allows the operations of multiple threads to be completed in a single update of a shared object. By adding operation combining to the small object protocol, a thread's operation may complete even though the thread never successfully updated the object itself. Adding operation combining to the small object protocol requires two auxiliary structures: an *announcement* table and a *result* table, each of which have one entry per thread. In the wait-free protocol, a thread first writes its intended operation in the announcement table and then makes a local copy of the object's state and the result table. The thread then applies to its local copy of the object its own operation as well as all of the other operations listed in the announcement table. The results of these operations are stored in its local copy of the result table. After applying the operations, the thread attempts to install its local copy of the object and the result table using a *store-conditional*. If the update fails, the thread checks in the shared result table to see if another thread performed its operation already. If not, the thread loops back and tries the protocol again. Herlihy proves that a thread's operation is guaranteed to complete within two iterations of the protocol [Herlihy 91]. In my model, the wait free protocol incurs the same cost as the small

[1] I treat *load-linked* as a normal memory operation in the calculations.

object protocol, with the addition of $2N$ work due to the cost of copying of the result table and scanning of the announcement table.

7.4.4 Alemany and Felten's Solo Protocol

Both of Herlihy's protocols suffer from the problem that the repeated copying of the object performed by the unsuccessful threads slows down the progress of the successful thread. Alemany and Felten refer to this as *useless parallelism*, and they propose a way to reduce it using support from the operating system [Alemany & Felten 92]. To reduce useless parallelism, they develop a protocol that explicitly limits the number of concurrent threads to a small number K. When $K = 1$, they call this their *solo protocol*.

In the solo protocol, a shared counter is used to indicate the number of threads that are allowed to concurrently attempt an update to the object, and this counter is initially set to one. If a thread wants to update the shared object, it spins on the counter until its value is greater than zero. It then attempts to atomically decrement the counter's value if it is still greater than zero, and if it succeeds it applies Herlihy's small object protocol and atomically increments the counter. This protocol reduces useless parallelism in high contention cases since the majority of threads spin in the cache on the counter value rather than repeatedly copying the object over the interconnection network. In order to provide tolerance to delays, the operating system increments the counter when a thread blocks while updating the shared object, allowing another thread to attempt an update.

In the best case scenario, a thread incurs cost $C + 1$ to check and decrement the counter, cost $S + C + 3$ to apply the small object protocol, and cost C to increment the counter, for a total cost of $S + 3C + 4$. In the bad case scenario, the adversary can cause the counter to be a point of contention, forcing all N threads to read the counter and attempt to decrement it at cost $N(C+1)$. Adding this to the cost of the small object protocol and an increment yields a total cost of $N(C + 1) + S + 2C + 3$ for the bad base adversary. In the worst case scenario, the scheduler has the power to block threads, and this can be used to create even more work. In the worst case, the adversary schedules the update in the same way as the bad case with the exception that the thread that acquires the counter is blocked before it updates the pointer to the shared object. The operating system increments the counter and the

adversary lets another thread attempt its update, again maximizing work and again blocking the thread before it completes its update. This continues until all of the threads have been blocked in an update, at which time the adversary unblocks them all and lets them all attempt to install their version of the object. This creates work $\sum_{i=0}^{N} i(C+1) + S + 2C + 3 = N(S+2C+3) + \frac{N(N+1)}{2}(C+1)$ in the worst case.

7.4.5 Alemany and Felten's Solo Protocol with Logging

An additional inefficiency of Herlihy's small object protocol is that it copies the entire state of the object even if the update only changes a small percentage of the object's state. Since only one thread updates the shared object at a time in the solo protocol, it is safe to directly update the shared version of the object provided that there is a mechanism to back out a partial update if a threads blocks during its operation. In the solo protocol with logging, threads do not copy the object and instead update the shared version and log their changes. In the event that a thread blocks during an update, this log is used by the operating system to roll the shared object back to its original state, and the thread is required to begin its operation again when it unblocks. The solo protocol with logging incurs the same work in our model for the best and bad case as a spin-lock with the exception that writes to the object incur an extra cost of 1 per write in order to log the original value of the changed word of state.

While immune to deadlock, Alemany and Felten's solo protocol with logging can be made to livelock by the worst case adversary. In the solo protocol with logging, when a thread blocks, its partial operation is first undone by the operating system using the log, the thread is then removed from the critical section and another thread is allowed to begin its operation. By repeatedly allowing threads to make progress and then blocking them before completion the scheduler can cause the protocol to process indefinitely without completing an operation. Thus, while retaining many of the benefits of the lock-free protocols, Alemany and Felten's solo protocol with logging is neither wait-free nor non-blocking. This livelock behavior is not exclusive to an adversarial scheduler. If a thread's operation is too long to execute in a single scheduling quantum, the thread will always block in the operation, and this will cause livelock.

7.4.6 Barnes's Caching Method

The final existing lock-free protocol I evaluate is *Barnes's caching method* [Barnes 93]. Of all of the black-box protocols, only Barnes's caching method produces implementations that allow threads to make progress in parallel. Barnes's protocol is extremely complicated and I do not attempt to fully explain it here. Instead, I give a high-level description of how it works and the cost it incurs in my model.

In Barnes protocol, threads first load the necessary data and perform their operations on the local cached copy at cost $(R+W)$. In order to cooperate safely, Barnes's protocol effectively creates a program that the threads interpret together. The program is first installed at a cost of C. The program is then interpreted at a cost of $(4C+2)$ per operation read and $(6C+3)$ per operation write. An optimization can be made for the first read or write resulting in a reduction of $(2C+1)$. This totals $R(4C+3)+W(6C+4)-C-1$. In the worst and bad scenarios, all N threads can be made to perform all $R(4C+3)+W(6C+4)-C-1$ work.

7.4.7 The solo-cooperative protocol

Choosing among the existing synchronization protocols involves a tradeoff between theoretical progress guarantees and practical performance. At one extreme, Herlihy's wait-free protocol offers the strongest progress guarantees and incurs substantial overhead in doing so. Herlihy's small object protocol gives up the wait-free property for the non-blocking property and a decrease in latency. Similarly, Barnes's caching method gives up the wait-free property in exchange for non-blocking and parallelism which could possibly improve performance. Alemany and Felten's solo protocol performs better under contention but is not robust to processor failure. Their solo protocol with logging offers even lower latency but gives up non-blocking and is simply tolerant of common thread delays. Lastly, there are finely tuned locks that offer good performance but no tolerance of delays.

No protocol exists that offers the same performance as locks and tolerance to some delays. Because most existing protocols have evolved from the more theoretical protocols, it is not surprising to find that such a protocol does not exist. I now describe a protocol that offers good performance in practice and is insensitive to preemption delays. In order to achieve this, I combine the idea of Barnes-style thread cooperation with the single active thread used in Alemany and Felten's solo protocol.

```
Concurrent_Apply(op_name, op_args) {
   repeat {
      repeat {
         old_op_ptr := op_ptr;
      }until (old_op_ptr <> 0);
   }until (Compare&Swap(op_ptr, old_op_ptr, 0) = Success);

   if (old_op_ptr <> 1) {
      Complete_Partial_Operation(old_op_ptr);
      }
   ret_val := Sequential_Apply(op_name, op_args);
   op_ptr := 1;
   return(ret_val);
}
```

Figure 7.1: Pseudocode for main body of the solo-cooperative protocol.

The main problem with thread cooperation is that large overhead is introduced when protecting the threads from each other. By having only one thread active at a time, however, cooperation can be achieved with no overhead in the common case of an operation that does not block. In the *solo-cooperative* protocol, a single thread at a time is allowed to update the shared object, similar to locking. This single thread updates the object without making a copy and without logging the changes being made. In order to provide insensitivity to delays, I include a mechanism that allows waiting threads to help finish the work of a blocked thread. If a thread blocks during an operation, its state is stored in the object, and its ownership of the object is released. A new thread can then use the stored state to finish the partially completed operation and begin to apply its own changes. Anderson uses a technique similar to this in the protection of his user-level scheduler in his scheduler activations work [Anderson et al. 92].

Like Alemany and Felten's protocols, I rely on support from the operating system

to execute code on behalf of a thread when it blocks and unblocks. Support of this kind is offered by extensible systems such as Spin [Bershad et al. 94a] and by kernel mechanisms like scheduler activations [Anderson et al. 92].

Pseudocode for the body of the protocol appears in Figure 7.1. In the solo-cooperative protocol, the object is protected by a variable called *op_ptr* which is functionally similar to a lock. If *op_ptr* is 0, the object is "locked"; if *op_ptr* is 1 the object is free; and for all other values, the object is free, but there is a partially finished operation to be finished, and *op_ptr* points its description. To update the shared object, a thread first reads *op_ptr* and *Compare&Swap*s 0 for a non-zero value, guaranteeing exclusive access. If the thread sees that the old value of *op_ptr* is 1, the thread applies its operation, sets *op_ptr* back to 1 and returns. If the old value of *op_ptr* is not 1, the thread completes the partially finished operation and subsequently applies its own operation.

When a thread blocks during an operation, the system bundles up the operation's state (stored in the thread's stack and registers) and stores a pointer to this in *op_ptr*. This has the effect of releasing *op_ptr*, thus making additional updates to the shared object possible even though the executing thread has blocked.

When a thread that blocked during an operation is unblocking, it checks to see if its operation has already been completed by another thread. If the waking thread's operation has not been completed, it will re-acquire *op_ptr* and will finish its operation. If the waking thread's operation has been completed, it will read the operation's result from its control structure and will continue execution after the operation.

In the case that a new thread acquires *op_ptr* and finds a partially complete operation, it cooperates by loading the blocked thread's registers and continues the execution of the partial operation. On completion of the blocked thread's operation, the cooperating thread writes the operation's result in the blocked thread's control structure and returns to apply its own operation.

This solo-cooperative protocol has the problem that if a thread blocks on I/O, such as a page fault, all of the threads in the system that attempt to cooperate will also block on this fault. While this may seem like a major flaw, it may have little impact in practice. In general I expect contention for an object to be low and that there will be no waiting threads. In the case that there are waiting threads, there is a good chance that the waiting threads will also need data that is on the faulting

page. While it is possible that additional progress could be made which would go unexploited by this protocol, I expect that this would not happen sufficiently often to be a problem.

In the best and bad case scenarios, the solo-cooperative protocol executes the same number of non-local memory accesses and synchronization instructions as the spin-lock. The worst case adversary can cause the solo-cooperative protocol to livelock, similar to the solo protocol with logging. Thus this protocol is also neither wait-free nor non-blocking. Recall that in the protocol, the state of a blocked thread's partially completed operation is stored under *op_ptr*. Upon encountering a partially complete operation, a thread will try to load the state of this operation and finish it. To force livelock, the scheduler first allows a thread to partially complete an operation and then forces it to block. The scheduler can then repeatedly allow a new thread to begin loading the state of the partially completed operation, and then block it before it makes any progress on the operation, thus causing livelock. This behavior seems limited, however, to the strongest adversary and there is no reason to expect this to occur in practice.

7.5 Evaluation

From Table 7.1, we see that Herlihy's protocols and Alemany and Felten's solo protocol incur overhead S due to their need to copy the entire object. The overhead renders these protocols impractical for objects with a large amount of state, such as a thread stack. However, for objects with a small amount of state, such as a shared counter, these protocols should be competitive in practice.

In Table 7.1, we also see that Barnes' caching method relies heavily on synchronization instructions. Barnes claims that although a large amount of overhead is incurred per operation instruction, critical sections are designed to be short and simple. This technique could conceivably perform better than a copying protocol if the objects were large and the operations consisted of only a few instructions.

In the high-contention bad case, we see that large amounts of memory contention can be generated by Barnes's and Herlihy's protocols. In order to make these protocols viable in the presence of a large number of active threads, they need a concurrency restriction mechanism like exponential backoff. In the scenarios in which threads do not block, we see that Alemany and Felten's solo protocol, the solo-cooperative

protocol and the spin-lock provide the combination of low latency and low contention. Interestingly, these are the same three protocols that are neither non-blocking nor wait-free, and all three can be kept from making progress by the worst case adversary.

7.6 Validation

The goal of this performance model is to make the evaluation of protocol performance easy and fairly accurate. In order to investigate its accuracy, I now compare the predictions of the bad case scenario with results produced by a parallel machine simulator for Herlihy's small object protocol and wait-free protocol, Alemany and Felten's solo protocol and solo protocol with logging, the solo-cooperative protocol, and for reference a *test-and-Compare&Swap* spin-lock. Although queue-based spin locks have some performance advantages over spin-locks, I chose a *test-and-Compare&Swap* spin-lock because of its simplicity and low latency. I did not simulate Barnes's caching protocol due to its implementation complexity. In order to obtain information about execution performance, I ran the protocols in Proteus, a parallel machine simulator developed at MIT [Brewer et al. 92]. Proteus is an execution driven simulator that takes as input augmented C programs and outputs execution times from a simulated parallel run of the program.

For the simulations, Proteus was configured to model a bus-based multiprocessor running the Goodman cache-coherency protocol [Goodman 83]. In the Proteus simulator, cache misses were 6 times as expensive as local cached instructions in the absence of contention, and synchronization instructions were 12 times as expensive ($C = 2$). During the simulations, each thread ran on its own processor, and thread delays did not occur. Thus the special-case code for the protocols using operating system support was never executed. The simulations consisted of a number of threads, varying from 1 to 20, alternately updating a shared object and performing 200 instructions worth of local work. I simulated the protocols using both a shared counter and a circular queue as the shared object.

In the simulations, I did not include exponential backoff for any of the protocols. Exponential backoff reduces resource contention by reducing concurrency, but does not change the relationship between concurrency and contention in a given protocol. Given that I am focusing on the interaction between concurrency and resource contention, there was no additional insight to be gained by including exponential

backoff.

7.6.1 Converting Work to Time

Recall that the bad scenario predicts the maximum amount of *work* that the adversary can arrange to take place for a single update, while the parallel machine simulations output execution times. In order to compare the model predictions to the simulation results, the work measure from the model needs to be converted to a time measure.

Let the amount of work predicted by the bad case adversary for a particular protocol in Table 7.1 be $W(N)$. Let L represent the time it takes a thread to perform the 200 instructions of local work. Since we are considering a bus based machine and since we only count operations that require the bus in our model, I assume that all of the synchronization protocol work is forced to serialize. I also assume that the local work does not need the bus and can be executed in parallel with other work. Given that I have allowed the bad case adversary to schedule threads to maximize work, I also allow the adversary to interleave local work with shared object updates in order to maximize time as well.

Herlihy's wait-free protocol is different that the others in that the cooperation provided by operation combining allows the operations of multiple threads to complete with a single update to the shared object. For now, consider the other protocols that do not exhibit this behavior. To maximize time, the adversarial scheduler allows one thread's update to occur at a time cost of $W(N)$. Before this thread can start its local work, it allows another thread to update the object, and so on. Once all N threads have updated the object, the adversary lets all N threads perform their local work in parallel. In this way, the adversary can force N updates to take time $NW(N) + L$. This results in a predicted time cost of $W(N) + \frac{L}{N}$ per operation for Herlihy's small object protocol, Alemany and Felten's protocols, the solo-cooperative protocol, and the *test-and-Compare&Swap* spin-lock.

Herlihy's wait-free protocol uses operation combining which allows the operations of multiple threads to complete with a single update. I make the assumption that since the adversary is forcing the maximum amount of work to occur, the maximum benefits of operation combining are realized, and N operations complete in a single update. Thus for Herlihy's wait-free protocol, the scheduler arranges for N operations to take time $W(N)$ and then schedules the local work at a cost of L. This yields an

average time cost per operation of $\frac{W(N)}{N} + \frac{L}{N}$. In the next section I compare these time predictions with simulation results to investigate the accuracy of the performance model.

7.6.2 Shared Counter

In the first set of simulations, threads alternately update a shared counter and perform local work until a total of 4096 updates have been done. Figure 7.2 shows the number of simulated cycles per update, varying the number of threads from 1 to 20. The graph shows that initially, as the parallelism in the system is exploited, additional processors improve performance. As the number of processors further increases, however, performance degrades as memory contention increases due to synchronization overhead. The small object protocol suffers large update times due to the contention caused by the constant copying. The solo protocol performs significantly better than the small object protocol, demonstrating the effectiveness of its concurrency limiting mechanism. The solo-cooperative protocol and the spin lock perform almost exactly the same in this simulated execution. This graph also clearly shows the amortizing effect that operation combining has on Herlihy's wait-free protocol. While the wait-free protocol is much slower initially, the cooperating has a beneficial effect when contention is high.

For the shared counter, the model variables are $S = 1$, $R = 0$, $W = 1$. The 200 instructions of local work have a time cost of $L = 200/6 = 33.3$. A graph of the predicted time per operation from the model is given in Figure 7.3. This graph correctly predicts the high memory contention generated by the small object protocol. It predicts reduced contention for the solo protocol and still less contention for the solo protocol with logging. It also correctly predicts the similar performance of the solo-cooperative protocol and the spin lock. The curve for Herlihy's wait-free protocol is similar in shape to the simulations, although the crossovers occur in different places.

These two graphs are surprisingly similar considering the basis of our model. The simulation is modeling a real machine which should behave in a stochastic manner. Our model is predicting worst case behavior for a powerful adversary. The fact that these two graphs are comparable indicates that a consistent percentage of the worst case contention actually occurs in practice.

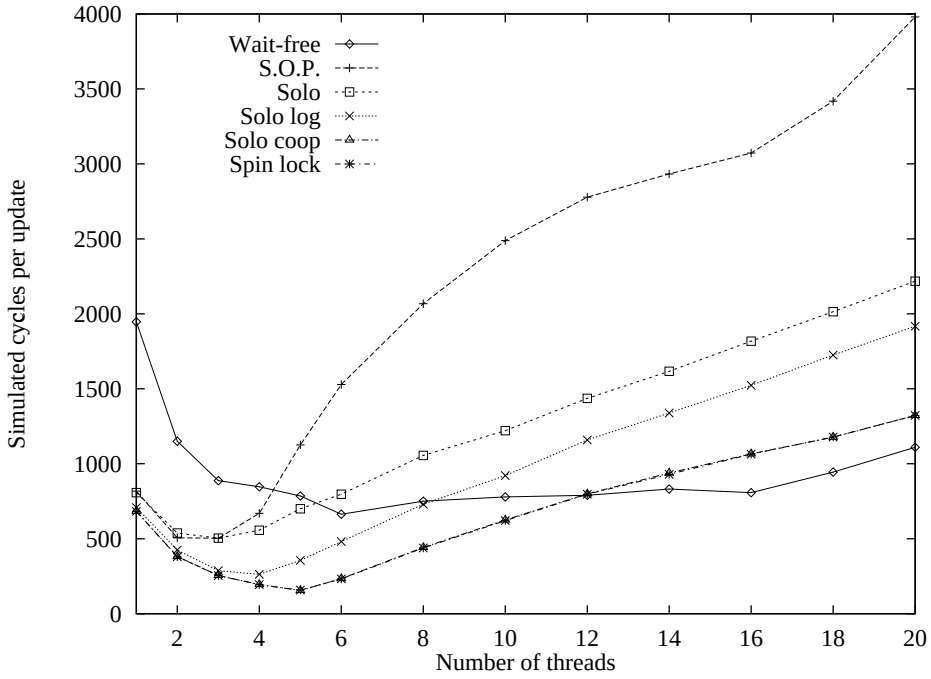

Figure 7.2: Simulated shared counter.

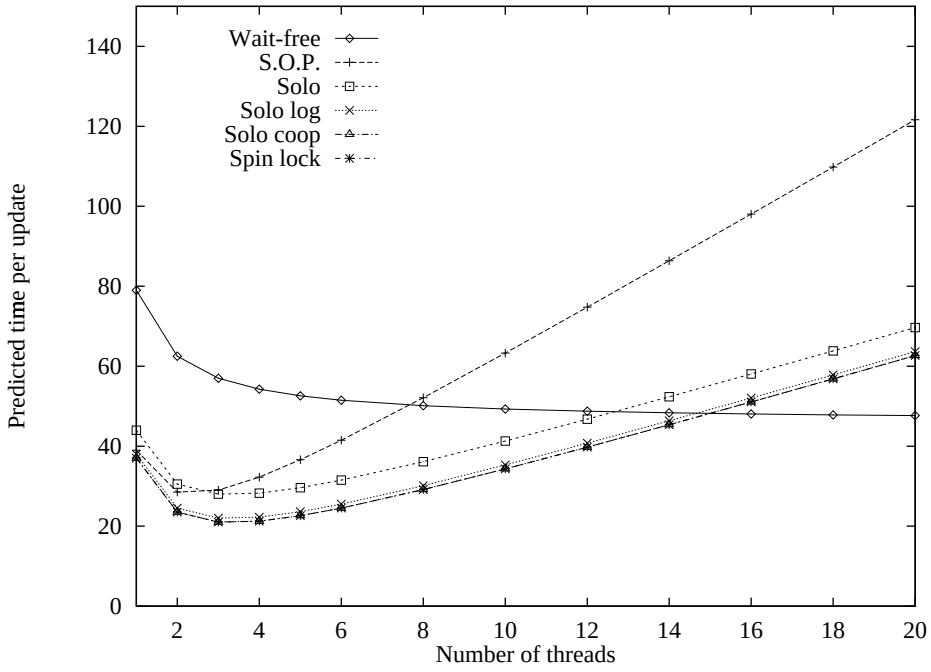

Figure 7.3: Predicted shared counter.

7.6.3 Circular Queue

In the second set of simulations, threads alternately enqueue and dequeue elements from a 64 entry circular queue and then perform local work. As before, the threads perform a total of 4096 updates per run, and the number of threads is varied from 1 to 20. Figure 7.4 shows the results for the simulations. This graph shows that both of Herlihy's protocols and the solo protocol incur significantly more overhead than the others. This overhead is due to the larger object size and shows why these protocols are only practically viable for small concurrent objects. The three non-copying protocols perform similarly, with the logging overhead of the solo protocol with logging separating it slightly from the solo-cooperative protocol and the spin lock.

The circular queue has model variables $S = 9$ and $R = 1$, $W = 3$ for enqueue and $R = 2$, $W = 2$ for dequeue. Since roughly the same number of enqueues and dequeues occur, I model the structure as having $S = 9$, $R = 1.5$ and $W = 2.5$. Local work was again set to $L = 33.3$. Figure 7.5 shows the results predicted by my model. The model predictions are relatively accurate for the non-copying protocols. The model correctly predicts the high contention for the small object protocol. For the solo protocol, however, the model underestimates the amount of contention that occurred in the simulation. The model also predicts that the wait-free protocol would outperform the copying protocols at high contention, which it did not. This is likely due to our optimistic assumption that N operations complete per update.

Despite these shortcomings, these graphs suggest that the model captures the important performance characteristics of lock-free synchronization protocols. The model is simple to apply and should serve as a useful tool for those designing and analyzing lock-free synchronization algorithms.

7.7 Summary

This chapter investigates the performance of lock-free synchronization protocols. A simple performance model is developed which is based solely on the cost of cache misses and synchronization instructions. In order to show the impact that these costs have on overall performance, the predictions of the model are compared with the results of parallel machine simulations. Despite the simplicity of the model,

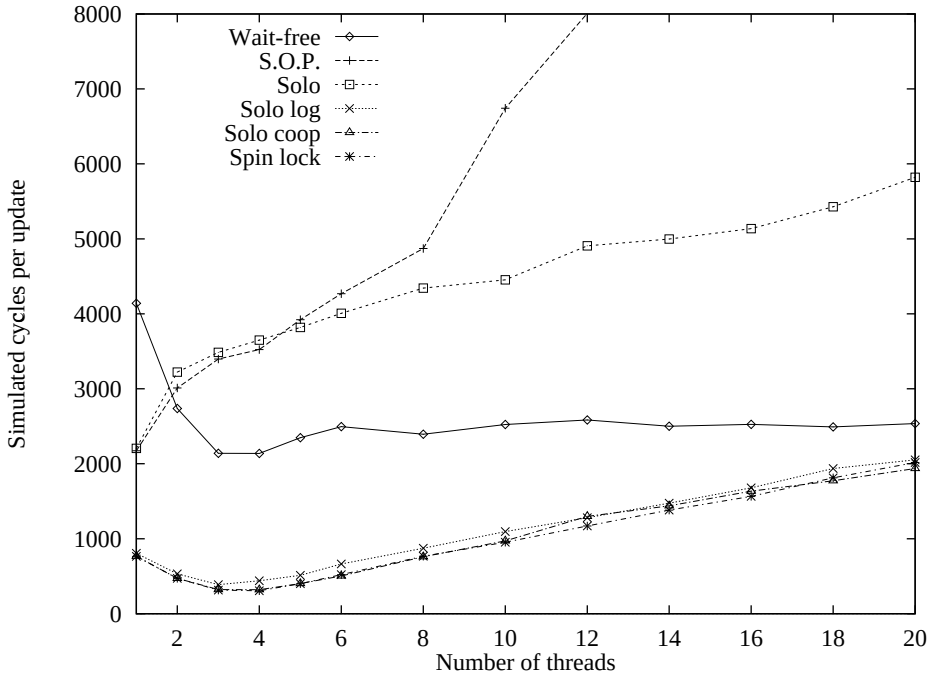

Figure 7.4: Simulated circular queue.

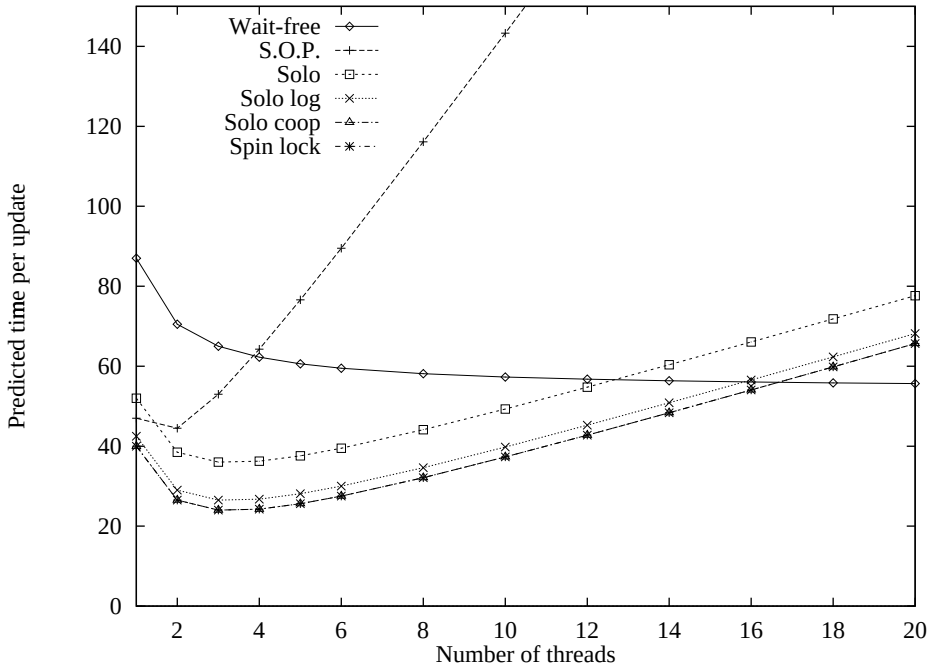

Figure 7.5: Predicted circular queue.

it does a fair job of predicting the relative performance of the protocols. One of the goals of this thesis is to show the importance of memory systems performance in algorithm design. Previous chapters have demonstrated the large impact that memory system performance has on the overall performance of sequential algorithms. The comparisons in this chapter show that in parallel environments as well, good overall performance cannot be achieved without good cache performance.

The analyses of the existing protocols in the model indicate that none of them offers insensitivity to delays as well as the same performance as locks in the common case. Accordingly, this chapter also presents a new lock-free synchronization protocol based on the combination of two existing protocols. The new protocol offers the same good performance as locks yet offers insensitivity to thread preemption delays.

Chapter 8

Conclusions

This thesis investigates the design and analysis of algorithms in the presence of caching. Since the introduction of caches, miss penalties have been steadily increasing relative to cycle times and have grown to the point where good performance cannot be achieved without good cache performance. As a result, a number of areas including compilers, runtime systems and operating systems account for the effects that caches have on overall performance. Unfortunately, the algorithm design community has largely ignored caches and their impact on performance.

The reluctance of this community to incorporate caching into their models is understandable. Caches have traditionally been transparent to the programmer, considered by most to be a hidden mechanism that speeds up memory accesses. The cache constants such as capacity and miss penalty are seldom exported to the programmer, discouraging optimizations which require this information. Caches also complicate traditional unit-cost performance models. When caching is taken into account, memory accesses no longer have unit-cost but instead incur a cost dependent on the pattern of previous references. Despite these complications, cache miss penalties have grown to the point that algorithm designers can no longer ignore the interaction between caches and algorithms.

To show the necessity of this paradigm shift, a large part of this thesis is devoted to demonstrating the potential performance gains of cache-conscious design. Efficient implementations of classic searching and sorting algorithms are examined for inefficiencies in their memory behavior, and simple memory optimizations are applied to them. The performance results demonstrate that these memory optimizations significantly reduce cache misses and improve overall performance. Reductions in cache

misses range from 40% to 90%, and although these reductions come with an increase in instruction count, they translate into execution time speedups of up to a factor of two. These results show that algorithm performance indeed obeys Amdahl's law. As cache miss penalties continue to increase, algorithms spend a larger and larger percentage of their time servicing cache misses. As a result, the potential impact of memory optimizations is growing larger relative to the impact of reductions in instruction count. In addition to sequential algorithms, this result also holds for parallel algorithms, despite differences in the memory systems of sequential and parallel machines. To demonstrate that the design principles that apply to sequential algorithms also apply to parallel algorithms, this thesis presents a performance evaluation of lock-free synchronization protocols. The study shows that like sequential algorithms, the overall performance of parallel algorithms is significantly affected by their cache performance.

In this thesis, a number of memory optimizations are applied to algorithms in order to improve their overall performance. What follows is a summary of the design principles developed in this work:

- Improving cache performance even at the cost of an increased instruction count can improve overall performance.

- Knowing and using architectural constants such as cache size and block size can improve an algorithm's memory system performance beyond that of a generic algorithm.

- Developing a more compact representation of a structure reduces its footprint in memory which can significantly improve cache performance.

- Reducing conflict misses improves the cache performance of the algorithm being optimized as well as the other algorithms with which it interacts.

- Spatial locality can be improved by adjusting an algorithm's structure to fully utilize cache blocks.

- Temporal locality can be improved by padding and adjusting data layout so that structures are aligned within cache blocks.

- Capacity misses can be reduced by processing large data sets in cache-sized pieces.

- Conflict misses can be reduced by processing data in cache-block-sized pieces.

Since the need for cache-conscious algorithm design is new, it is not surprising that there is a lack of analytical tools to help algorithm designers understand the memory behavior of algorithms. This thesis investigates techniques for analyzing the cache performance of algorithms. To explore the feasibility of a purely analytical technique, this thesis introduces *collective analysis*, a framework within which cache performance can be predicted as a function of both cache and algorithm configuration. Collective analysis uses a fairly simplistic model which assumes that algorithms can be accurately modeled with independent memoryless processes. This assumption reduces its general appeal as few algorithms fit this model well. Nevertheless, collective analysis is performed on implicit heaps with excellent results. The intuition provided by the analysis led to an important optimization and the performance predictions are surprisingly accurate considering the simplicity of the analysis. These results suggest that a more general model would be a useful tool for understanding the cache performance of algorithms. To make collective analysis more generally applicable, processes need to be extended to model more realistic memory reference patterns such as sequential traversals and bursty groups of references. This would complicate the calculation of the expected hit intensity of a process within a region. The analysis would have to accurately predict the interactions between an arbitrary number of processes exhibiting any of the modeled reference patterns, but the resulting framework would be applicable to a much larger class of algorithms.

The general lesson that can be learned from this thesis is that while computers generally follow the basic von Neumann model on which unit-cost models are based, there are implementation artifacts that significantly impact performance. Caches are one such artifact. This thesis has demonstrated the importance of caching in both algorithm design and analysis. The generalization of this is that these implementation artifacts need to be accounted for in analysis techniques if performance is to be well understood and in design techniques if the best possible performance is to be achieved.

Bibliography

[Agarwal et al. 89] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

[Agarwal et al. 94] R. Agarwal, F. Gustavson, and M. Zubair. Exploiting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM Journal of Research and Development*, 38(5):563–576, Sep 1994.

[Aggarwal & Vitter 88] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[Aggarwal et al. 87a] A. Aggarwal, K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *28th Annual IEEE Symposium on Foundations of Computer Science*, pages 204–216, 1987.

[Aggarwal et al. 87b] A. Aggarwal, K. Chandra, and M. Snir. A model for hierarchical memory. In *19th Annual ACM Symposium on Theory of Computing*, pages 305–314, 1987.

[Aho & Ullman 83] E. Aho, A. Hopcroft and D. Ullman. *Data structures and algorithms*. Addison-Wesley, Reading, MA, 1983.

[Alemany & Felten 92] J. Alemany and E. W. Felten. Performance Issues in Non-blocking Synchronization on Shared-Memory Multiprocessors. In *11th Annual ACM Symposium on Principles of Distributed Computing*, August 1992.

[Alpern et al. 94] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3):72–109, 1994.

[Anderson & Moir 95] J. H. Anderson and M. Moir. Universal constructions for large objects. In *DISTALG*, pages 168–182, September 1995.

[Anderson & Woll 91] R. J. Anderson and H. Woll. Wait-Free Parallel Algorithms for the Union-Find Pproblem. In *23st Annual ACM Symposium on Theory of Computing*, pages 360–370, 1991.

[Anderson 90] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[Anderson et al. 92] T. E. Anderson, B. N. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallel ism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[Barnes 93] G. Barnes. A method for implementing lock-free data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms & Architecture*, June 1993.

[Bershad 93] B. N. Bershad. Practical Considerations for Non-Blocking Concurrent Objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–274, May 1993.

[Bershad et al. 92] B. N. Bershad, D. Redell, and J. Ellis. Fast Mutual Exclusion for Uniprocessors. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–233, October 1992.

[Bershad et al. 93] B. N. Bershad, M. Zekauskas, and W. Sawdon. The Midway Distributed Shared Memory System. In *Proceedings of the 38th IEEE Computer Society International Conference (COMPCON '93)*, pages 528–537, February 1993.

[Bershad et al. 94a] B. N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. SPIN–An Extensible Microkernel for Application-specific Operating System Services. Technical Report UW-CSE-94-03-03, Department of Computer Science and Engineering, University of Washington, February 1994.

[Bershad et al. 94b] B. N. Bershad, D. Lee, T. Romer, and J. B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–70, 1994.

[Blelloch et al. 91] G. Blelloch, C. Plaxton, C. Leiserson, S. Smith, B. Maggs, and M. Zagha. A comparison of sorting algorithms for the connection machine. In *Proceedings of the 3rd ACM Symposium on Parallel Algorithms & Architecture*, pages 3–16, July 1991.

[Brewer et al. 92] E. A. Brewer, A. Colbrook, C. N. Dellarocas, and W. E. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. In *1992 ACM Sigmetrics*, pages 247–8, June 1992.

[Carlsson 91] S. Carlsson. An optimal algorithm for deleting the root of a heap. *Information Processing Letters*, 37(2):117–120, 1991.

[Carr et al. 94] S. Carr, K. McKinley, and C. W. Tseng. Compiler optimizations for improving data locality. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, 1994.

[Carter et al. 91] J. B. Carter, J. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[Cierniak & Li 95] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of the 1995 ACM Symposium on Programming Languages Design and Implementation*, pages 205–217. ACM, 1995.

[Clark 83] D. Clark. Cache performance of the VAX-11/780. *ACM Transactions on Computer Systems*, 1(1):24–37, 1983.

[Coffman & Denning 73] E. Coffman and P. Denning. *Operating Systems Theory*. Prentice–Hall, Englewood Cliffs, NJ, 1973.

[Cormen et al. 90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[Culler et al. 93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

[De Graffe & Kosters 92] J. De Graffe and W. Kosters. Expected heights in heaps. *BIT*, 32(4):570–579, 1992.

[Dig 92] Digital Equipment Corporation. *DECchip 21064-AA Microprocessor, Hardware Reference Manual*, 1992. Order Number: EC-N0079-72.

[Diwan et al. 94] A. Diwan, D. Tarditi, and E. Moss. Memory subsystem performance of programs using copying garbage collection. In *Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, 1994.

[Doberkat 81] E. Doberkat. Inserting a new element into a heap. *BIT*, 21:225–269, 1981.

[Doberkat 82] E. Doberkat. Deleting the root of a heap. *Acta Informatica*, 17:245–265, 1982.

[Dongarra et al. 90] J. Dongarra, O. Brewer, J. Kohl, and S. Fineberg. A tool to aid in the design, implementation, and understanding of matrix algorithms for parallel processors. *Journal of Parallel and Distributed Computing*, 9(2):185–202, June 1990.

[Feller 71] W. Feller. *An introduction to probability theory and its applications.* Wiley, New York, NY, 1971.

[Fenwick et al. 95] D. Fenwick, D. Foley, W. Gist, S. VanDoren, and D. Wissell. The AlphaServer 8000 series: High-end server platform development. *Digital Technical Journal*, 7(1):43–65, 1995.

[Floyd 64] R. W. Floyd. Treesort 3. *Communications of the ACM*, 7(12):701, 1964.

[Fortune & Wyllie 78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[Gannon et al. 88] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, Oct 1988.

[Gonnet & Munro 86] G. Gonnet and J. Munro. Heaps on heaps. *SIAM Journal of Computing*, 15(4):964–971, 1986.

[Goodman 83] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pages 124–131, June 1983.

[Gotlieb 81] L. Gotlieb. Optimal multi-way search trees. *SIAM Journal of Computing*, 10(3):422–433, Aug 1981.

[Graunke & Thakkar 90] G. Graunke and S. Thakkar. Synchronization Alorithms for Shared-Memory Multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[Grunwald et al. 93] D. Grunwald, B. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In *Proceedings of the 1993 ACM Symposium on Programming Languages Design and Implementation*, pages 177–186. ACM, 1993.

[Hennesey & Patterson 90] J. Hennesey and D. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufman Publishers, Inc., San Mateo, CA, 1990.

[Herlihy & Moss 93] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. *20th Annual International Symposium on Computer Architecture*, 21(2):289–300, May 1993.

[Herlihy 90] M. Herlihy. A Methodology for Implementing Highly Concurrent Structures. *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, March 1990.

[Herlihy 91] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects., 1991. CRL Technical Report 91/10.

[Hill & Smith 89] M. Hill and A. Smith. Evaluating associativity in CPU caches. *ACM Transactions on Computer Systems*, 38(12):1612–1630, 1989.

[Hoare 62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[Holberton 52] F. E. Holberton. In *Symosium on Automatic Programming*, pages 34–39, 1952.

[Horowitz et al. 96] M. Horowitz, M. Mortonosi, T. Mowry, and M. D. Smith. Informing memory operations: Providing memory performance feedback in modern processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[Huang & Langston 88] B. C. Huang and M. A. Langston. Practical in-place merging. *Communications of the ACM*, 31(3):348–352, March 1988.

[Hui & Sevcik 94] L. Hui and K. C. Sevcik. Parallel sorting by overpartitioning. In *Proceedings of the 6th ACM Symposium on Parallel Algorithms & Architecture*, pages 46–56, June 1994.

[Hwu & Chang 89] W. Hwu and P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 242–251, May 1989.

[Johnson 75] D. B. Johnson. Priority queues with update and finding minimum spanning trees. *Information Processing Letters*, 4, 1975.

[Johnson 95] T. Johnson. Characterizing the performance of algorithms for lock-free objects. *IEEE Transactions on Computers*, 44(10):1194–1207, Oct 1995.

[Jones 86] D. Jones. An emperical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, 1986.

[Kennedy & McKinley 92] K. Kennedy and K. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 International Conference on Supercomputing*, pages 323–334, 1992.

[Kessler & Hill 92] R. Kessler and M. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):338–359, Nov 1992.

[Knuth 73] D. E. Knuth. *The Art of Computer Programming, vol III – Sorting and Searching.* Addison–Wesely, Reading, MA, 1973.

[Koopman et al. 92] P. Koopman, P. Lee, and P. Siewiorek. Cache behavior of combinator graph reduction. *ACM Transactions on Programming Languages and Systems*, 14(2):265–97, April 1992.

[Lam et al. 89] T. Lam, P. Tiwari, and M. Tompa. Tradeoffs between communication and space. In *21st Annual ACM Symposium on Theory of Computing*, pages 217–226, 1989.

[LaMarca 94] A. LaMarca. A performance evaluation of lock-free synchronization protocols. In *13th Annual ACM Symposium on Principles of Distributed Computing*, pages 130–140, August 1994.

[Lebeck & Wood 94] A. Lebeck and D. Wood. Cache profiling and the spec benchmarks: a case study. *Computer*, 27(10):15–26, Oct 1994.

[Lenoski et al. 92] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–105, May 1992.

[Lowney et al. 93] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. ODonnell, and J. Ruttenberg. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7:51–142, 1993.

[Manber 89] U. Manber. *Introduction to algorithms : a creative approach.* Addison-Wesley, Reading, MA, 1989.

[Martonosi et al. 95] M. Martonosi, A. Gupta, and T. Anderson. Tuning memory performance of sequential and parallel programs. *Computer*, 28(4):32–40, 1995.

[Massalin & Pu 89] H. Massalin and C. Pu. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the 12th Symposium on Operating Systems Principles*, pages 191–200, December 1989.

[Mattson et al. 70] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 7(1):78–117, 1970.

[Mellor-Crummey & Scott 91] J. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1), February 1991.

[Naor et al. 91] D. Naor, C. Martel, and N. Matloff. Performance of priority queue structures in a virtual memory environment. *Computer Journal*, 34(5):428–437, Oct 1991.

[Rao 78] G. Rao. Performance analysis of cache memories. *Journal of the ACM*, 25(3):378–395, 1978.

[Reingold & Wilfred 86] E. Reingold and H. Wilfred. *Data structures in Pascal.* Little, Brown, Boston, MA, 1986.

[Sedgewick 77] R. Sedgewick. The analysis of quicksort programs. *Acta Informatica*, 7:327–355, 1977.

[Sedgewick 78] R. Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, October 1978.

[Sedgewick 88] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, MA, 1988.

[Singh et al. 92] J. Singh, H. Stone, and D. Thiebaut. A model of workloads and its use in miss-rate prediction for fully associative caches. *IEEE Transactions on Computers*, 41(7):811–825, 1992.

[Singleton 69] R. C. Singleton. An efficient algorithm for sorting with minimal starage: Algorithm 347. *Communications of the ACM*, 12(3):185–187, March 1969.

[Sleator & Tarjan 85] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[Smith 85] A. J. Smith. Cache evaluation and the impact of workload choice. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 63–73, June 1985.

[Snyder 86] L. Snyder. Type Architecture, Shared Memory and the Corollary of Moest Potential. *Annual Review of Computer Science*, 1:289–318, 1986.

[Srivastava & Eustace 94] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the 1994 ACM Symposium on Programming Languages Design and Implementation*, pages 196–205. ACM, 1994.

[Taylor et al. 90] G. Taylor, P. Davies, and M. Farmwald. The TBL slice:a low-cost high-speed address translation mechanism. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 355–363, 1990.

[Temam et al. 94] O. Temam, C. Fricker, and W. Jalby. Cache interference phenomena. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 261–271, 1994.

[Temam et al. 95] O. Temam, C. Fricker, and W. Jalby. Influence of cross-interfernces on blocked loops: A case study with matrix-vector multiply. *ACM Transactions on Programming Languages and Systems*, 17(4):561–575, 1995.

[Torrellas et al. 92] J. Torrellas, A. Gupta, and J. Hennessy. Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 162–174, October 1992.

[Uhlig et al. 94] R. Uhlig, D. Nagle, T. Stanley, T. Mudge, S. Sechrest, and R. Brown. Design tradeoffs for software-managed TLBs. *ACM Transactions on Computer Systems*, 12(3):175–205, 1994.

[Valois 95] J. D. Valois. Lock-free linked lists using compare-and-swap. In *14th Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, August 1995.

[Verkamo 88] A. Verkamo. External quicksort. *Performance Evaluation*, 8(4):271–288, Aug 1988.

[Verkamo 89] A. Verkamo. Performance comparison of distributive and mergesort as external sorting algorithms. *The Journal of Systems and Software*, 10(3):187–200, Oct 1989.

[Wegner & Teuhola 89] L. Wegner and J. Teuhola. The external heapsort. *The Journal of Systems and Software*, 15(7):917–925, Jul 1989.

[Weiss 95] M. Weiss. *Data structures and algorithm analysis.* Benjamin/Cummings Pub. Co., Redwood City, CA, 1995.

[Welbon et al. 94] E. Welbon, C. Chan-Nui, D. Shippy, and D. Hicks. The power2 performance monitor. *IBM Journal of Research and Development*, 38(5):545–554, Sep 1994.

[Williams 64] J. W. Williams. Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.

[Wing & Gong 90] J. M. Wing and C. Gong. A Library of Concurrent Objects and Their Proofs of Correctness., 1990. Technical Report CMU-CS-90-151, Carnegie Mellon University.

[Wing & Gong 93] J. M. Wing and C. Gong. Testing and Verifying Concurrent Objects. *Journal of Parallel and Distributed Computing*, 17(2):164–182, February 1993.

[Wolf & Lam 91] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the 1991 ACM Symposium on Programming Languages Design and Implementation*, pages 30–44. ACM, 1991.

[Wolfe 89] M. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, pages 655–664, 1989.

[Zahorjan et al. 88] J. Zahorjan, E. Lazowska, and D. Eager. Spinning Verses Waiting in Parallel Systems with Uncertainty. In *Proceedings of the International Seminar on Distributed and Parallel Systems*, pages 455–472, December 1988.

Vita

Anthony G. LaMarca was born in New York, New York on August 9, 1968. He received the Bachelor of Arts degree in Computer Science, with highest distinction, from the University of California at Berkeley in 1989. From 1990 to 1996, he attended the University of Washington, receiving his M.S. degree in Computer Science and Engineering in 1992, and his Ph.D. in Computer Science and Engineering in 1996.