# *An algorithmic theory of caches*

by

Sridhar Ramachandran

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY.

December 1999

© Massachusetts Institute of Technology 1999.
All rights reserved.

Author ______________________________

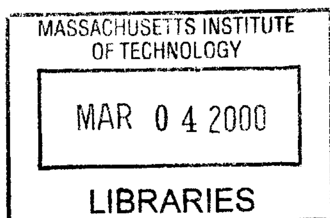
Department of Electrical Engineering and Computer Science
Jan 31, 1999

Certified by ______________________________

Charles E. Leiserson
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ______________________________

Arthur C. Smith
Chairman, Departmental Committee on Graduate Students



ENG

# *An algorithmic theory of caches*

by

Sridhar Ramachandran

*Submitted to the*
*Department of Electrical Engineering and Computer Science*
*on Jan 31, 1999 in partial fulfillment of the*
*requirements for the degree of Master of Science.*

### *Abstract*

The ideal-cache model, an extension of the RAM model, evaluates the referential locality exhibited by algorithms. The ideal-cache model is characterized by two parameters—the cache size $Z$, and line length $L$. As suggested by its name, the ideal-cache model practices automatic, optimal, omniscient replacement algorithm. The performance of an algorithm on the ideal-cache model consists of two measures—the RAM running time, called work complexity, and the number of misses on the ideal cache, called cache complexity.

This thesis proposes the ideal-cache model as a "bridging" model for caches in the sense proposed by Valiant [49]. A bridging model for caches serves two purposes. It can be viewed as a hardware "ideal" that influences cache design. On the other hand, it can be used as a powerful tool to design cache-efficient algorithms. This thesis justifies the proposal of the ideal-cache model as a bridging model for caches by presenting theoretically sound caching mechanisms closely emulating the ideal-cache model and by presenting portable cache-efficient algorithms, called cache-oblivious algorithms.

In Chapter 2, we consider a class of caches, called random-hashed caches, which have perfectly random line-placement functions. We shall look at analytical tools to compute the expected conflict-miss overhead for random-hashed caches, with respect to fully associative LRU caches. Specifically, we obtain good upper bounds on the conflict-miss overhead for random-hashed set-associative caches. We shall then consider the augmentation of fully associative victim caches [33], and find out that conflict-miss overhead reduces dramatically for well-sized victim caches. An interesting contribution of Chapter 2 is that victim caches need not be fully associative. Random-hashed set-associative victim caches perform nearly as well as fully associative victim caches.

Chapter 3 introduces the notion of cache-obliviousness. Cache-oblivious algorithms

are algorithms that do not require knowledge of the parameters of the ideal cache, namely $Z$ and $L$. A cache-oblivious algorithm is said to be optimal if it has asymptotically optimal work and cache complexity, when compared to the best cache-aware algorithm, on any ideal-cache. Chapter 3 describes optimal cache-oblivious algorithms for matrix transposition, FFT, and sorting. For an ideal cache with $Z = \Omega(L^2)$, the number of cache misses for an $m \times n$ matrix transpose is $\Theta(1 + mn/L)$. The number of cache misses for either an $n$-point FFT or the sorting of $n$ numbers is $\Theta(1 + (n/L)(1 + \log_Z n))$. Chapter 3 also proposes a $\Theta(mnp)$-work algorithm to multiply an $m \times n$ matrix by an $n \times p$ matrix that incurs $\Theta(1 + (mn + np + mp)/L + mnp/L\sqrt{Z})$ cache faults. All the proposed algorithms rely on recursion, suggesting divide-and-conquer as a powerful paradigm for cache-oblivious algorithm design.

In Chapter 4, we shall see that optimal cache-oblivious algorithms, satisfying the "regularity condition," perform optimally on many platforms, such as multilevel memory hierarchies and probabilistic LRU caches. Moreover optimal cache-oblivious algorithms can also be ported to some existing manual-replacement cache models, such as SUMH [8] and HMM [4], while maintaining optimality.

Thesis Supervisor: Charles E. Leiserson
Title: Professor of Computer Science and Engineering

# *Acknowledgments*

I thank my advisor Charles E. Leiserson for his invaluable guidance. I thank Harald Prokop for introducing me to his work on cache-oblivious algorithms. I enjoyed working with him on this subject. I am grateful to Matteo Frigo and Derek Chiou for their contribution to this thesis.

I owe a lot to the research environment at LCS. I thank Phil Lisiecki, Bin Song, Dimitris Mitsouras, Nitin Thaper, and Don Dailey for making Supertech an interesting group. Special thanks to Alejandro Caro, Jan-Willem Maessen, James Hoe, and Sandeep Chatterjee.

I would like to thank my parents for their support throughout my stay at Boston.

# Contents

**1 Introduction** **9**

1.1 Ideal-cache model: a bridging model for caches . . . . . 10

1.2 Contributions of the thesis . . . . . 13

**2 Building the impossible: an ideal cache** **17**

2.1 Related work . . . . . 19

2.2 Probabilistic LRU caches . . . . . 22

2.3 Balls and bins . . . . . 26

2.4 Victim caches . . . . . 29

2.5 Random-hashed victim caches . . . . . 34

2.6 Simulation results . . . . . 36

2.7 Conclusion . . . . . 38

**3 Cache-oblivious algorithms** **43**

3.1 Matrix multiplication . . . . . 46

3.2 Matrix transposition and FFT . . . . . 49

3.3 Funnelsort . . . . . 53

3.4 Distribution sort . . . . . 58

3.5 Conclusion . . . . . 62

**4 Cache-oblivious algorithms on other cache models** **65**

4.1 Multilevel caches . . . . . 66

4.2 Probabilistic LRU caches . . . . . 68

4.3 Memories with manual replacement: SUMH and HMM . . . . . 69

4.4 Conclusion . . . . . 74

**5 Conclusion** **77**

**A Bibliograpy** **81**

CHAPTER 1

# Introduction

As processor clock rates increase rapidly, computer throughput is increasingly limited by memory bandwidth [41]. Main memory is typically much slower than the CPU. To improve memory performance, computer architects add one or more levels of caches to the memory hierarchy. Caches are small, fast memories close to the CPU which hold items that the CPU is likely to access. Today, almost all computers include one or more levels of on-chip or off-chip caches [28].

Among all mechanisms to improve computer performance, caches are arguably the simplest and most effective. Extensive research has been done in academia and industry to build faster, bigger, more concurrent, and smarter caches. Simulation studies of caching mechanisms on memory traces generated by programs are the most popular and widely accepted method of analyzing caches. Few theoretically sound caching mechanisms have been implemented, however. Consequently, a need exists for a theoretical framework to evaluate the worth of caching mechanisms.

While designing caches, computer architects tacitly assume that memory accesses made by programs exhibit spatial and temporal locality. Unfortunately, most algorithms today are engineered for efficient execution in flat memory models, such as the RAM [7] model. These algorithms do not generally exhibit much locality of reference. Consequently, a need exists for caching models that evaluate and reward referential locality displayed by algorithms.

This thesis proposes an extension of the RAM model, called the ideal-cache model, as a "bridging" model for caches. Section 1.1 elaborates the notion of a bridging model, and advocates the ideal-cache model as a bridging model for caches. The rest of the thesis justifies this proposal, as summarized in Section 1.2, by providing theoretically sound caching mechanisms and portable cache-efficient algorithms.

## 1.1 *Ideal-cache model: a bridging model for caches*

This thesis advocates the ideal-cache model as a bridging model for caches. In this section, we discuss the what a bridging model for caches is, and we review related work in modeling caches. This section then presents the ideal-cache model—an automatic, fully associative cache model with optimal replacement.

#### *What is a bridging model?*

The notion of a bridging model is inspired by Valiant's bridging model for parallel computation [49]. Valiant's bridging model, called the BSP (Bulk-Synchronous Parallel) model, aims at providing an abstraction for parallel computers. Algorithm designers optimize their algorithms for the BSP model, while computer architects improve the BSP parameters of their parallel computers.

In a similar vein, a bridging model for caches, as illustrated in Figure 1-1, provides a framework for algorithm and cache design. In this thesis, the ideal-cache model is proposed as a bridging model for caches. Algorithm designers develop algorithms that work efficiently on the ideal-cache model. On the other hand, computer architects design memory hierarchies that perform nearly as well as the ideal cache.

#### *Traditional cache models*

Before looking at the ideal-cache model, we shall review previous work in cache modeling. The traditional approach to modeling caches is to propose a parameterized model capturing the nonuniform memory access cost in memory hierarchies. HMM [4], BT [5], and UMH [8] are examples of such parameterized models. An algorithm is optimized to work efficiently for the parameters of the memory hierarchy. Cache designers are expected to improve the parameters for their caches.

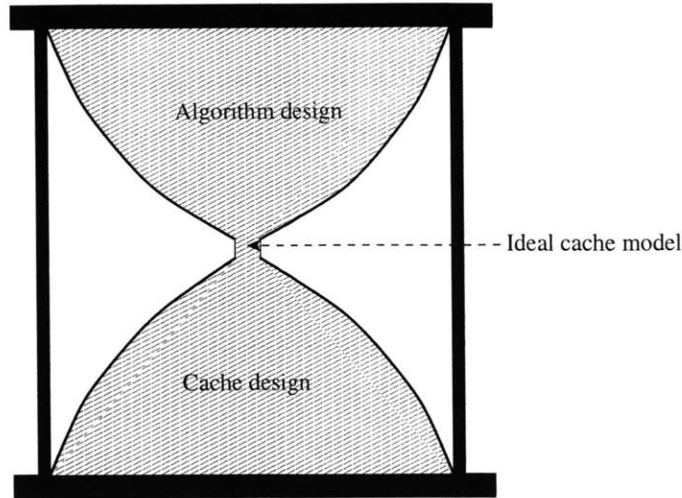



**Figure 1-1:** Bridging model for algorithm and cache design.

| Processor | L1 | L2 |
|---|---|---|
| Pentium III | 16K/16K 4-way associative | 256K 8-way associative |
| AMD K6-III | 32K/32K 2-way associative | 256K 4-way associative |
| HP PA-8500 | 512K/1M 4-way associative | — |
| UltraSparc 2 | 8K/8K direct-mapped | 512K direct-mapped |
| Alpha 21164 | 8K/8K direct-mapped | 96K 3-way associative |

**Figure 1-2:** Memory hierarchies of typical modern processors.

Many traditional models suffer from a drawback, however. Traditional cache models typically assume software control over caches [4, 5, 8]. On the other hand, most practical caches are transparent to the programmer [32, Section 2.4.2]. In fact, a programmer does not have total control over caches in any of the architectures shown in Figure 1-2.

Given the legacy of automatically managed caches, we can appreciate that software cache control only augments, but does not replace, automatic cache management. I believe that providing complete software control over caches may result in complexity that hardware designers may choose to avoid. Recently, the Computer-Aided Automation Group at MIT Laboratory for Computer Science, started the Malleable Caches project [17] to research limited cache-control directives.

The drawback of designing algorithms for cache models that assume software-managed caches is twofold. Firstly, the algorithm designer must manage the memory hierarchy

manually. According to my experience, efficient manual management of a multilevel memory hierarchy is a strenuous exercise. Secondly, efficient algorithms developed for software-managed cache models cannot necessarily be easily ported to typical memory hierarchies that are automatically managed. Consequently, the assumption of software-managed caches degrades the usefulness of a cache model.

The assumption of manual memory management is reasonable for out-of-core memory models, however. Using software environments such as TPIE [21], a programmer can exert complete control over external memory. Shriver and Nodine [43], and Vitter [50] show that memory models with explicit management aid the development of efficient external memory algorithms.

### *Ideal-cache model*

The ideal-cache model is a simple, two-level cache model with automatic replacement, that evaluates the performance of algorithms using two complexity measures—the work complexity and the cache complexity. We now look at the details of the ideal-cache model.

The ideal-cache model, which is illustrated in Figure 1-3, consists of a computer with a two-level memory hierarchy consisting of an ideal (data) cache of $Z$ words, and an arbitrarily large main memory. Because the actual size of words in a computer is typically a small, fixed size (4 bytes, 8 bytes, etc.), we shall assume that word size is constant; the particular constant does not affect our asymptotic analyses. The cache is partitioned into ***cache lines***, each consisting of $L$ consecutive words which are always moved together between cache and main memory. Cache designers typically use $L > 1$, banking on spatial locality to amortize the overhead of moving the cache line. We shall generally assume in this thesis that the cache is ***tall***:

$$Z = \Omega(L^2) , \tag{1.1}$$

which is usually true in practice.

The processor can only reference words that reside in the cache. If the referenced word belongs to a line already in cache, a ***cache hit*** occurs, and the word is delivered to the processor. Otherwise, a ***cache miss*** occurs, and the line is fetched into the cache. The ideal cache is ***fully associative*** [28, Ch. 5]: cache lines can be stored anywhere in the cache. If the cache is full, a cache line must be evicted. The ideal cache uses the optimal off-line

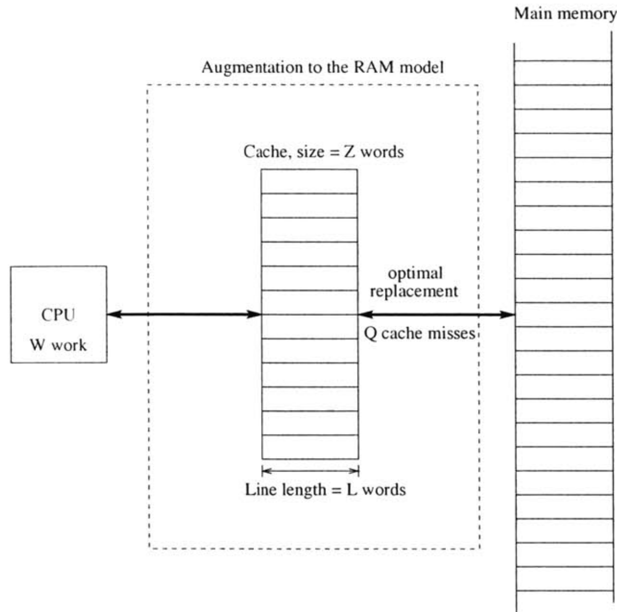



**Figure 1-3:** The ideal-cache model.

strategy of replacing the cache line whose next access is furthest in the future [10], and thus it exploits temporal locality perfectly.

Unlike various other hierarchical-memory models [4, 5, 8, 11] in which algorithms are analyzed in terms of a single measure, the ideal-cache model uses two measures. An algorithm with an input of size $n$ is measured by its ***work complexity*** $W(n)$—its conventional running time in a RAM model [7]—and its ***cache complexity*** $Q(n; Z, L)$—the number of cache misses it incurs as a function of the size $Z$ and line length $L$ of the ideal cache. When $Z$ and $L$ are clear from context, we denote the cache complexity simply as $Q(n)$ to ease notation.

## 1.2 *Contributions of the thesis*

The rest of this thesis justifies the ideal-cache model as an apt bridging model for caches. In this section, we look at various issues involved in adopting the bridging framework, and a brief outline of results addressing these issues. The contribution of this thesis can be broadly characterized into three categories. The first category of results, presented in Chapter 2, relates to the design of theoretically sound caching mechanisms closely emulating the ideal-cache model. Results in this category justify the ideal-cache model as a hard-

ware "ideal" influencing cache design. The second category of results, presented in Chapter 3, relates to the design of portable cache-efficient algorithms, called cache-oblivious algorithms, that perform well on any ideal cache without any tuning whatsoever. Results in this category justify the ideal-cache model as a powerful model for algorithm design. The third category of results, presented in Chapter 4, relates to the portability of cache-oblivious algorithms to more complicated cache models—such as multilevel caches and some traditional cache models that assume software-controlled caches—without loss of efficiency.

***Ideal-cache model: a hardware ideal***

Optimal replacement depends on the future accesses made by the algorithm [10]. Since cache line replacement is an online problem, ideal caches cannot be implemented in practice. We therefore address the question,

> *How can we design caches that exhibit near-optimal performance?*

in Chapter 2. Sleator and Tarjan [45] show that fully associative LRU caches are asymptotically competitive to proportionally smaller ideal caches. Unfortunately, fully associative caches do not have efficient practical implementations [28, pg. 568]. Cache designers limit the associativity of caches to achieve fast lookup. Typical set-associative caches have small associativity, as shown in Figure 1-2. Limiting cache associativity creates conflict misses [30], which are also called interference misses. The conflict-miss overhead for set-associative caches has been studied extensively using simulation [29, 40, 41], and to a restricted extent, using analytical techniques [2]. Recently, pseudorandom line placement was proposed as a mechanism to reduce conflict misses [27]. In Chapter 2, we consider a class of caches, called random-hashed caches, that have perfectly random line-placement functions. We shall look at analytical tools to compute the expected conflict-miss overhead for random-hashed caches, with respect to fully associative LRU caches. Specifically, we obtain good upper bounds on the conflict-miss overhead for random-hashed set-associative caches. We shall then consider the augmentation of fully associative victim caches [33], and find out that conflict-miss overhead reduces dramatically for well-sized victim caches. An interesting contribution of Chapter 2 is that victim caches need not be fully associative. Random-hashed set-associative victim caches perform nearly as well as

fully associative victim caches. To summarize, Chapter 2 justifies the ideal-cache model as a hardware ideal by suggesting practical designs for near-ideal caches.

### *Ideal-cache model: a powerful model for algorithm design*

Usually, algorithms that perform efficiently on a parameterized cache model require the knowledge of the cache parameters. We refer to such algorithms as ***cache-aware*** algorithms. The ideal-cache model has two parameters, namely $Z$ and $L$. A cache-aware algorithm requires the values of these parameters to tune itself. In this thesis, we shall focus on ***cache-oblivious algorithms***, that do not require the parameters $Z$ and $L$. In other words, cache-oblivious algorithms exhibit good performance on all ideal caches. A cache-oblivious algorithm is said to be ***optimal*** if it has asymptotically optimal work and cache complexity, when compared to the best cache-aware algorithm, on any ideal-cache. Being a subset of cache-aware algorithms, optimal cache-oblivious algorithms may be harder, if not impossible to design. We therefore address the following questions:

*How can optimal cache-oblivious algorithms be designed?*

*What are the applicable paradigms?*

*Do all problems permit optimal cache-oblivious algorithms?*

Chapter 3, which presents research done jointly with Matteo Frigo, Charles Leiserson and Harald Prokop, describes optimal cache-oblivious algorithms for matrix transposition, FFT, and sorting. For an ideal cache with $Z = \Omega(L^2)$, the number of cache misses for an $m \times n$ matrix transpose is $\Theta(1 + mn/L)$. The number of cache misses for either an $n$-point FFT or the sorting of $n$ numbers is $\Theta(1 + (n/L)(1 + \log_Z n))$. I shall also propose a $\Theta(mnp)$-work algorithm to multiply an $m \times n$ matrix by an $n \times p$ matrix that incurs $\Theta(1 + (mn + np + mp)/L + mnp/L\sqrt{Z})$ cache faults. All the proposed algorithms rely on recursion, suggesting divide-and-conquer as a powerful paradigm for cache-oblivious algorithm design. For the last question regarding the power of cache oblivious algorithms, however, existing insight seems discouraging. Bilardi and Peserico [11] show that some computation DAGs do not permit optimal cache-oblivious execution in the HRAM model. Peserico believes that this result can be extended to the ideal-cache model [38]. I have still not encountered a practical problem that is not solvable by an optimal cache-oblivious algorithm, however. The results presented in Chapter 3, I feel, justify the ideal-cache model

as a powerful model for algorithm design.

### *Porting cache-oblivious algorithms*

The ideal-cache model assumes a perhaps-questionable and overly simplified cost model. We therefore address the question,

*How do optimal cache-oblivious algorithms perform on complex memory hierarchies?*

In Chapter 4, which presents research done jointly with Matteo Frigo, Charles Leiserson and Harald Prokop, we shall see that optimal cache-oblivious algorithms on ideal caches perform optimally on many platforms. These platforms include multilevel memory hierarchies and probabilistic LRU caches. Optimal cache-oblivious algorithms can also be ported to some existing manual-replacement cache models, such as SUMH [8] and HMM [4], while maintaining optimality. We shall also look at some automated memory management routines that enable porting.

Chapter 5 contains my concluding remarks on this thesis along with open problems that interest me. Two interesting open questions are the following:

*Do the caching mechanisms proposed in this thesis have practical significance?*

*How do cache-oblivious algorithms perform on real computers with caches?*

We shall look at preliminary results and related work addressing these questions in Chapter 5.

CHAPTER 2

# *Building the impossible: an ideal cache*

Building an ideal cache is impossible, since optimal replacement cannot be implemented in an online setting such as caches [10]. Cache designers should, therefore, be satisfied with caches that exhibit near-optimal performance. This chapter makes a strong case for caches with random line-placement functions by proving that such caches are nearly ideal.

Fully associative LRU [28] caches are provably comparable to proportionally smaller ideal caches [45]. Unfortunately, searching an item in a fully associative cache consumes too much time, severely limiting their practical worth [28, pg. 568]. Cache designers prefer set-associative caches due to their faster search times [30, 41]. A ***line-placement function*** $H$ is used to map a memory address to a set in the cache, as shown in Figure 2-1. Conflict misses [30] occur when too many data items map to the same set, causing potentially reusable cache lines in the set to be flushed out, despite sufficient capacity in the overall cache. Typically, the line-placement function is a simple modulo function implemented by extracting bits from the memory address. Some computations are significantly slowed down in set-associative caches having this naive line-placement function, however [25, 27]. Intuitively, we can see that conflict misses are largely due to bad line placement rather than low associativity. In this chapter, we theoretically and using simulation prove this intuition. Specifically, we analyze caches with perfectly random line-placement functions,

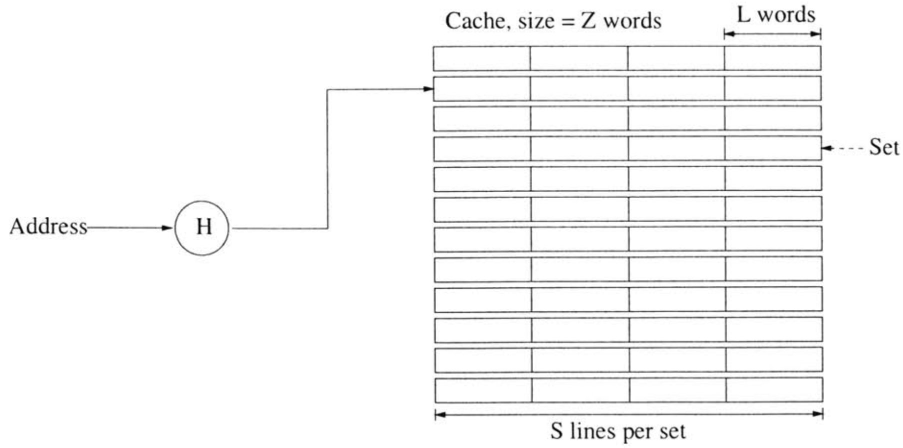



**Figure 2-1:** Set-associative cache with line-placement function $H$.

called ***random-hashed caches***. We bound the "conflict-miss overhead," which is the fraction of memory accesses that are conflict misses, for random-hashed set-associative caches. Cache designers add victim caches [33] to reduce conflict misses. In this chapter, we analyze the benefits of adding victim caches, theoretically. If the victim cache is capable of holding the expected number of "overflowing lines," the conflict-miss overhead is reduced by a factor exponential in the size of the victim cache. We also look at random-hashed set-associative caches as a scalable alternative to proportionally smaller fully associative victim caches.

We now look at the notation used and assumptions made throughout the chapter. A set-associative cache with a capacity of $Z$ words and $S$-way associativity, such as that shown in Figure 2-1, is partitioned into cache lines holding $L$ words, each of which belongs to a set holding $S$ cache lines. We are interested in caches with small associativity $S$. The number of cache lines in the cache is denoted by $n_L = Z/L$. The total number of sets in the cache is denoted by $n_S = Z/SL$. Cache lines in a set are maintained using LRU replacement. Since at most $S$ cache lines belong to a set, maintaining LRU information takes only $\lceil \lg S \rceil$ bits. Hwang and Briggs [32, pg. 116] present some possible designs for small LRU-managed sets. The main memory is also divided into lines of $L$ contiguous words. Each line is associated with a line address. Each set is assigned to a set address ranging from 0 to $n_S - 1$. The line-placement function

$$H : \{0,1,2,\ldots\} \to \{0,\ldots,n_S - 1\}$$

maps a given line address to a set. We assume that the placement function $H$ is a perfectly

random hash function, which satisfies the distribution

$$\Pr\{(H(i_1)=j_1)\wedge\cdots\wedge(H(i_k)=j_k)\}=1/n_S^k$$

for any $k>0$, valid line addresses $i_1,\ldots,i_k$, and valid set addresses $j_1,\ldots,j_k$.

The rest of this chapter is as follows. We first discuss previous work related to this chapter in Section 2.1. Section 2.2 presents an interesting class of randomized caches called probabilistic LRU caches. An "absence" function $P$ completely characterizes a probabilistic LRU cache, also called, a $P$-LRU cache when the absence function $P$ is known. Section 2.2 proves that the expected number of cache misses of a $P$-LRU cache has a conflict-miss overhead of $P(i)$ when compared to a fully associative cache with $i$ cache lines. Probabilistic LRU caches are valuable since random-hashed set-associative caches are $P$-LRU caches for a computable absence function $P$. Section 2.3 deals with obtaining good upper bounds and approximations for the absence function using the well-studied balls and bins problem [36, Section 3.1]. Random-hashed set-associative caches augmented with victim caches [33] are also probabilistic LRU caches. Section 2.4 evaluates the absence function for such caches. In Section 2.5, we consider victim caches that are not fully associative. Set-associative victim caches with random-hashed line placement can replace proportionally smaller fully associative victim caches without considerable degradation in performance. Section 2.6 presents simulation results predicting the expected cache misses incurred by some SPEC95 floating point benchmarks on random-hashed caches. These results, plotted in Figures 2-9 through 2-15, show that caches with small associativity can exhibit provably close performance to fully associative LRU caches, and that small victim caches can remarkably reduce the miss rate. Section 2.7 presents my concluding remarks on this chapter relating my work to previous and ongoing research on cache design.

## 2.1 *Related work*

The replacement algorithm used by a cache heavily influences its miss rate. Many researchers have analyzed various cache replacement algorithms theoretically. Belady [10] proved that the best replacement strategy is to replace the cache line accessed farthest in the future. Due to its dependence on the future of the memory-access sequence, Belady's algorithm cannot be used in an online setting.

Since optimal cache replacement is impossible, theoreticians have pursued algorithms that could nearly match the performance of ideal caches.

**Definition 1** *A cache replacement strategy is said to be $\eta$-competitive if there exists a constant $b$, such that, on every sequence of requests, the number $Q$ of cache misses is at most $\eta Q^* + b$, where $Q^*$ is the number of cache misses on an optimal cache.*

In 1985, Sleator and Tarjan [45] proved that an LRU cache with $k$ lines is $k/(k-k^*+1)$-competitive to an optimal cache with $k^*$ lines. They also proved that better bounds cannot be achieved by deterministic caching algorithms. If we compare LRU and ideal caches of the same size, the competitive ratio is

$$\begin{aligned} \frac{k}{k-k^*+1} &= \frac{k}{k-k+1} \\ &= k \end{aligned}$$

which is not impressive. If we compare LRU caches with smaller ideal caches, however, the competitiveness result can be attractive.

**Lemma 2** *Consider an algorithm that causes $Q_{\text{ideal}}(n;Z,L)$ cache misses on a problem of size $n$ using a $(Z,L)$ ideal cache. Then, the same algorithm incurs $Q_{\text{LRU}}(n;Z,L) \le 2Q_{\text{ideal}}(n;Z/2,L)$ cache misses on a $(Z,L)$ cache that uses LRU replacement.*

*Proof.* The number $k$ of lines in the LRU cache is $k = Z/L$. The number $k^*$ of lines in the ideal cache is $k^* = Z/2L = k/2$. Therefore, the competitive ratio is

$$\begin{aligned} \frac{k}{k-k^*+1} &= \frac{2k^*}{k^*+1} \\ &< 2\,. \end{aligned}$$

Thus, from Definition 1, $Q_{\text{LRU}}(n;Z,L) \le 2Q_{\text{ideal}}(n;Z/2,L) + b$, where $b$ is a constant. If both caches are empty initially, the constant $b$ is 0 [45]. □

In other words, an LRU cache performs at most twice as many cache misses as an ideal cache half its size. The importance of this result lies in the fact that the analysis is independent of the access sequence. In practice, fully associative LRU caches perform much better than the theoretical bound [54]. To summarize, fully associative LRU caches excel from a theoretical and practical standpoint.

Apart from the miss rate, cache design involves other performance metrics such as hit time and miss penalty. The hit time for a fully associative caches is dominated by the time

to perform an associative search. In the late 1980's, the use of caches with limited associativity was justified on the basis of their superior hit times, although their miss rates were slightly inferior in simulations [29, 40]. The cost of associativity is high even in current practice [28].

If a modulo line-placement function is used, some programs experience inordinate conflict misses. A common class of such memory-access sequences is shown in [27, Section 3]. Even simple matrix computations can generate many conflict misses for certain dimensions of a matrix [25, Figure 4 and 5].

As early as 1982, Smith's survey on caches [46] mentions the use of pseudorandom line-placement functions to reduce conflict misses. At that time, the advantage of using such functions did not justify the additional complexity in the cache. Recently, interest has been revived in this area. Using simulation techniques, Gonález, Valero, Topham, and Parcerisa [27] extensively study the performance gains in using placement functions based on XOR and polynomial arithmetic [42].

The well-studied random-replacement caches [36] differ slightly from random-hashed caches. To locate an item with line address $x$, random-hashed caches search only the set $H(x)$. On the other hand, a random-replacement cache must perform an associative search. Consequently, random-hashed caches can have better hit times than random-replacement caches. There are differences in performance as well. Consider a memory-access sequence of the form $a, b, a, b, \ldots$, where $a$ and $b$ are two different line addresses. A random-hashed cache with associativity $S = 1$, places $a$ and $b$ in sets $H(a)$ and $H(b)$. Every access is a cache hit if $H(a) \neq H(b)$, and otherwise every access is a cache miss. A random-replacement cache evicts a random cache line and may incur both cache hits and cache misses in the sequence.

Numerous other techniques have been proposed to reduce conflict misses in direct-mapped and set-associative caches. Jouppi [33, 34] proposed the addition of a victim cache to hold items that are evicted from the limited-associativity cache. Hash-rehash caches [1] and column-associative caches [3] use two hashing functions to avoid conflict misses.

| Time | Item accessed | Items ordered by $\text{rank}_t$ | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 |
| $t = 1$ | a | – | – | – | – |
| $t = 2$ | b | a | – | – | – |
| $t = 3$ | c | b | a | – | – |
| $t = 4$ | b | c | b | a | – |
| $t = 5$ | a | b | c | a | – |
| $t = 6$ | d | a | b | c | – |
| $t = 7$ | a | d | a | b | c |

**Figure 2-2:** Example showing the rank of items in a fully associative LRU cache.

## 2.2 *Probabilistic LRU caches*

In this section, we describe probabilistic LRU caches, an interesting class of randomized caches. A probabilistic LRU cache can be fully characterized by its absence function. Random-hashed caches discussed in this thesis belong to the class of probabilistic caches. We shall analyze the properties of probabilistic LRU caches in this section. Later, in Sections 2.3, 2.4, and 2.5, we shall show that some random-hashed caches are probabilistic LRU caches, and we shall obtain approximations and bounds on their absence functions.

We first look at fully associative LRU caching in more detail. Let us consider the cache misses made by algorithm $\mathcal{A}$. During the execution of algorithm $\mathcal{A}$, we can associate each line in the memory with a ***rank*** that gives the number of other *distinct* cache lines accessed after its most-recent access. The rank of a line $x$ before the $t$th memory operation is given by $\text{rank}_t(x)$. On the first access of a line $x$ in $\mathcal{A}$, its rank is $\infty$, since the most-recent access of $x$ is undefined. Figure 2-2 shows the rank of items for a sample memory-access sequence. In the example, the rank of item c at time $t = 4$ is 2. Items not displayed in the table have rank $\infty$.

**Lemma 3** *Let line $x$ be accessed by the $t$th memory operation of algorithm $\mathcal{A}$. Then, line $x$ is held in a fully associative LRU cache capable of holding $i$ cache lines if and only if* $\text{rank}_t(x) < i$.

*Proof.* A fully associative LRU cache maintains the $i$ most-recently accessed cache lines.

The lemma follows from the definition of rank. □

In order to simplify the forthcoming discussion, we define a function $\mathrm{V}al : \{false, true\} \to \{0,1\}$ as

$$\mathrm{V}al(A) = \begin{cases} 1 & \text{if } A \text{ is true,} \\ 0 & \text{otherwise.} \end{cases}$$

to simplify forthcoming mathematical presentation. The following corollary to Lemma 3 expresses the number of cache misses incurred on a fully associative cache using the $\mathrm{V}al$ function.

**Corollary 4** *A memory-access sequence $x_1, \ldots, x_\tau$ incurs*

$$Q_{\mathrm{LRU}}(i) \quad = \quad \sum_{t=1}^{\tau} \mathrm{V}al(\mathrm{rank}_t(x_t) \geq i) \tag{2.1}$$

*cache misses on a fully associative LRU cache with i cache lines. Consequently, the function $Q_{\mathrm{LRU}}(i)$ is monotonically decreasing, i.e., $\tau = Q_{\mathrm{LRU}}(0) \geq Q_{\mathrm{LRU}}(1) \geq Q_{\mathrm{LRU}}(2) \geq \cdots$.* □

The characterization of fully associative LRU caches, as given by Lemma 3, can be extended to incorporate randomized caches.

**Definition 5** *Consider a memory-access sequence $x_1, \ldots, x_\tau$. Let $P : \{0,1,2,\ldots\} \to [0,1]$ be a monotonically increasing function. For all $t \geq 0$, a* ***probabilistic LRU cache with absence function*** $P$ *misses $x_t$ on the* $t$th *access with probability (exactly)*

$$1 - P(\mathrm{rank}_t(x_t)) \; .$$

Any embodiment of a probabilistic LRU cache should include a random process, except, of course, fully associative LRU caches. A fully associative LRU cache can be defined as a probabilistic LRU cache with absence function $P$ satisfying

$$P(i) = \mathrm{V}al(i < Z) \; ,$$

for all $i \geq 0$.

Another possible embodiment of a probabilistic LRU cache is as follows. Consider a cache capable of maintaining any set $\mathcal{S}$ of lines. We assume that the cache does not have any capacity constraint on the cardinality of set $\mathcal{S}$. Upon a cache hit, the set $\mathcal{S}$ remains unchanged. Upon a cache miss on line $x$, a coin having success probability $p$ is tossed. If

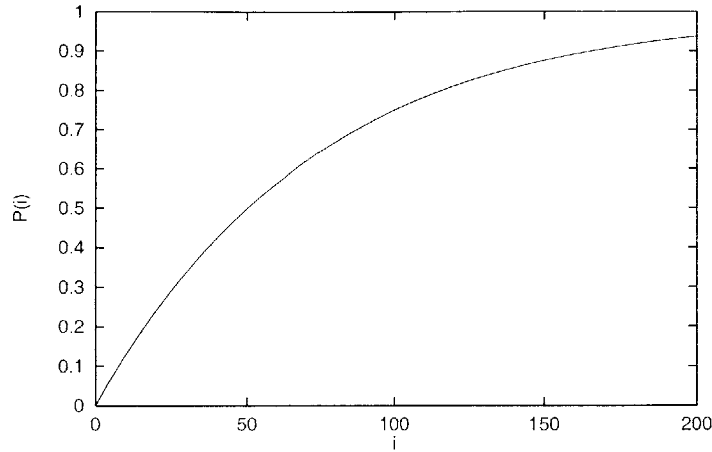



**Figure 2-3:** Graph showing an exemplary absence function $P$.

the coin toss is a success, line $x$ is added to the set, i.e., $\mathcal{S} \leftarrow \mathcal{S} \cup \{x\}$. Otherwise, the cache is flushed before incorporating $x$, i.e., $\mathcal{S} \leftarrow \{x\}$. Let us analyze the probability that the $t$th access in the memory-access sequence $x_1, \ldots, x_\tau$, is a cache hit. If $\text{rank}_t(x_t)$ is $\infty$, the cache misses on $x_t$, since $x_t$ has not been accessed previously. Otherwise, $\text{rank}_t(x_t)$ distinct lines are accessed after the most-recent access of $x_t$. For each distinct line accessed, the cache is not flushed with probability $p$. Therefore, the $t$th access is a cache hit with probability $p^{\text{rank}_t(x_t)}$. In other words, the cache under consideration is a $P$-LRU cache with absence function

$$P(i) = 1 - p^i ,$$

for all $i \geq 0$. Figure 2-3 shows the absence function of the above cache, for $p = 1/2^{1/50}$. According to the figure, an access made to a line after 50 other distinct lines have been accessed is a cache miss with probability exactly $1/2$. An access made to a line with rank 100, is a cache miss with probability exactly 0.75. Although probabilistic, a $P$-LRU cache exhibits very "predictable" behavior, since $P$ is a function associated with the cache and not the memory-access sequence.

The following lemma shows that the expected number of cache misses is a weighted sum of the cache misses incurred in fully associative LRU caches of all possible sizes.

**Lemma 6** *Let $Q_{\text{LRU}}(i)$ be the number of cache misses incurred by algorithm $\mathcal{A}$ on an LRU cache with i lines. Let $\tau$ be the number of memory operations performed by $\mathcal{A}$. Then, the expected number*

$\mathrm{E}[Q]$ *of cache misses is given by*

$$\mathrm{E}[Q] = \tau P(0) + \sum_{i>0} Q_{\mathrm{LRU}}(i)\,(P(i) - P(i-1))\,. \tag{2.2}$$

*Proof.* The expected number of cache misses is

$$\begin{aligned}
\mathrm{E}[Q] &= \sum_{t=1}^{\tau} P(\mathrm{rank}_t(x_t)) \\
&= \sum_{t=1}^{\tau} \left( P(0) + \sum_{i=1}^{\mathrm{rank}_t(x_t)} P(i) - P(i-1) \right) \\
&= \tau P(0) + \sum_{t=1}^{\tau} \sum_{i>0} \mathrm{V}al(\mathrm{rank}_t(x) \geq i)(P(i) - P(i-1)) \\
&= \tau P(0) + \sum_{i>0} (P(i) - P(i-1)) \sum_{t=1}^{\tau} \mathrm{V}al(\mathrm{rank}_t(x_t) \geq i) \\
&= \tau P(0) + \sum_{i>0} Q_{\mathrm{LRU}}(i)\,(P(i) - P(i-1))
\end{aligned}$$

from Equation (2.1) in Corollary 4. □

We now look at a rigorous definition of conflict-miss overhead.

**Definition 7** *Consider any memory-access sequence $x_1, \ldots, x_\tau$. Let $Q_{\mathrm{LRU}}$ be the number of cache misses on a fully associative LRU cache capable of holding $i$ lines. A random-hashed cache has* ***conflict-miss overhead*** *$\beta(i)$ if, for every memory-access sequence, the number $Q$ of cache misses is at most $Q_{\mathrm{LRU}} + \beta(i)\tau$.*

Let us consider an example illustrating Definition 7. Consider a random-hashed cache with expected conflict-miss overhead 0.01 with respect to a fully associative LRU cache of half the size. Compared to the fully associative LRU cache, the random-hashed cache incurs misses on an extra 1% of its memory accesses.

We now bound the conflict-miss overhead of probabilistic LRU caches.

**Theorem 8** *A P-LRU cache has an expected conflict-miss overhead $\beta(i) = P(i)$ with respect to a fully associative cache capable of holding $i$ cache lines.*

*Proof.* Let $\tau$ be the number of memory operations, and let $Q_{\mathrm{LRU}}(i)$ be the number of cache misses in a fully associative cache capable of holding $i$ cache lines. The expected number of cache misses according to Lemma 6 is

$$\mathrm{E}[Q] \;=\; \tau P(0) + \sum_{i>0} Q_{\mathrm{LRU}}(i)\,(P(i) - P(i-1))\,,$$

Definition 5 states that $P$ is a monotonically increasing function. Furthermore, we have

$$\tau \geq Q_{\mathrm{LRU}}(1) \geq Q_{\mathrm{LRU}}(2) \geq \cdots ,$$

according to Corollary 4. Thus, we can bound the expectation of the number of cache misses as

$$\begin{aligned} \mathrm{E}[Q] &\leq \tau P(Z) + \lim_{i\to\infty} Q_{\mathrm{LRU}}(Z)(P(i) - P(Z)) \\ &\leq \tau P(Z) + Q_{\mathrm{LRU}}(Z)(1 - P(Z)) \\ &\leq \tau P(Z) + Q_{\mathrm{LRU}}(Z), \end{aligned}$$

since $P(i) \leq 1$ for all $i$. □

Unlike Lemma 6, Theorem 8 makes no assumption about the $Q_{\mathrm{LRU}}$ function. In other words, Theorem 8 gives a "competitive" analysis of the conflict-miss overhead. Although the analysis made by Theorem 8 is not exact, the following lemma shows that it cannot be made any tighter.

**Lemma 9** *There exists a memory-access sequence that incurs an expected conflict-miss overhead of $P(i)$ in a P-LRU cache with respect to a fully associative cache capable of holding $i$ cache lines.*

*Proof.* Consider an infinite sequence of the form

$$a_1, a_2, \ldots, a_i,\ a_1, \ldots, a_i,\ a_1, \ldots ,$$

where each $a_j$ is distinct. Every element accessed after the first $i$ accesses has rank $i$. Therefore, the probabilistic LRU cache misses on these elements with probability $P(i)$. A fully associative cache capable of holding $i$ lines does not miss after the first $i$ accesses. Since we consider an infinite sequence, the conflict-miss overhead tends to $P(i)$. □

## 2.3 *Balls and bins*

In this section, we analyze the performance of random-hashed set-associative caches. First, we prove that random-hashed set-associative caches are probabilistic LRU caches having absence functions expressible as solutions to the balls and bins problem given in [36, Section 3.1]. Then, the absence function for random-hashed caches with small associativity is

approximated. Finally, Theorem 8 is used to evaluate conflict-miss overhead of random-hashed set-associative caches.

Recall that for set-associative caches, we defined parameters $n_L$ as the number of cache lines, $n_S$ as the number of lines in a set, and $S$ as the associativity. A random hash function $H$ is used to map the line address to a set address. Cache lines in a set are maintained using LRU replacement.

We now prove a crucial lemma showing that random-hashed caches are $P$-LRU caches for some absence function $P$. But first, we develop a notation to represent the balls-and-bins distribution.

**Definition 10** *Consider $i$ balls thrown uniformly at random into $n$ initially empty bins. The random variable $BB(i,n)$ denotes the number of balls in a given bin.*

**Lemma 11** *A random-hashed set-associative cache having $n_S$ sets of associativity $S$ is a probabilistic LRU cache. The absence function $P$ satisfies*

$$P(i) = \Pr\{BB(i,n_S) \geq S\} \ .$$

*Proof.* Consider a memory-access sequence $x_1, \ldots, x_T$. When a line $x_t$ is accessed by the $t$th operation, it is cached in set $H(x_t)$. Line $x_t$ can be evicted only by cache lines accessed after its most-recent access. From the definition of rank, we know that $\text{rank}_t(x_t)$ distinct lines have been accessed after the most-recent access of $x_t$. If less than $S$ of these lines map to $H(x_t)$, the access to $x_t$ hits the cache, an event which happens with probability $\Pr\{BB(i,n_S) < S\}$. Thus, the random-hashed cache is a $P$-LRU cache satisfying $P(i) = \Pr\{BB(i,n_S) \geq S\}$. □

**Corollary 12** *A random-hashed direct-mapped cache is a probabilistic LRU cache with absence function*

$$P(i) = 1 - (1 - 1/n_L)^i \ ,$$

*for all $i > 0$. The expected conflict-miss overhead such a cache with respect to a fully associative cache holding $i$ cache lines is $P(i)$.*

*Proof.* We use the definition of $P$ as in Lemma 11. The probability that at least one ball is present in a given bin after placing $i$ balls is given by

$$1 - (\Pr\{\text{a ball is not placed in the given bin}\})^i = 1 - (1 - 1/n_L)^i \ .$$

The rest of the lemma follows from Theorem 8. □

The analysis done in Corollary 12 can be extended to random-hashed caches with small set-associativity.

**Corollary 13** *A random-hashed cache having associativity $S \ll n_S$ is a P-LRU cache, where*

$$P(i) \approx 1 - e^{-i/n_S} \cdot \sum_{j=0}^{S-1} \frac{(i/n_S)^j}{j!} .$$

*Proof.* The probability of $j$ balls falling in a bin is

$$\begin{aligned} \binom{i}{j} \cdot \left(\frac{1}{n_S}\right)^j \left(1 - \frac{1}{n_S}\right)^{n_S - j} &\approx \frac{(i/n_S)^j}{j!} \left(1 - \frac{1}{n_S}\right)^{n_S - j} \\ &\approx \frac{(i/n_S)^j}{j!} e^{-i/n_S} , \end{aligned}$$

for $1 \le j \ll n_S$. The result follows from Lemma 11. □

For large $S$, we can use the Chernoff bound as follows.

**Corollary 14** *A random-hashed set-associative cache with $n_L$ cache lines and associativity $S$ is a P-LRU cache, where*

$$P(i) \le \left( \frac{e^{\delta/(1+\delta)}}{1+\delta} \right)^S ,$$

*for $\delta = n_L/i - 1$.*

*Proof.* The proof is similar to Corollary 12. The absence function is given by

$$\Pr\{\text{less than } S \text{ balls are placed in a given bin}\} .$$

Let indicator random variable $X_j$ be defined as

$$X_j = Val(\text{Ball } j \text{ is placed in the given bin}) .$$

The absence probability is given by

$$P(i) = \Pr\left\{ \sum_{j=0}^{i-1} X_j < S \right\} ,$$

which is a binomial random variable with expectation $i/n_S$. A binomial random variable $X$ with expectation $\mu$ satisfies the inequality

$$\Pr\{X \ge (1+\delta)\mu\} < \left( \frac{e^{\delta}}{(1+\delta)^{1+\delta}} \right)^{\mu} ,$$

according to the Chernoff bound given in [36, pg. 68]. We can apply this bound to yield

$$P(i) < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{i/n_S},$$

where $\delta = n_S S/i - 1 = n_L/i - 1$. □

Curiously, Corollaries 12, 13, and 14 relate $P(i)$ to functions of $S$ and $i/n_L$ that do not require the value of $i$. Corollary 13 gives good approximations for $P(i)$ function for typical values of $S$ and $n_L$. Figure 2-4 plots the function $P(i)$ versus $i/n_L$ for associativities $S = 1, 2, 3, 4, 8$, using Corollary 13. From Theorem 8, we have $\beta(i) = P(i)$. We can, therefore, interpret this graph as the conflict-miss overhead $\beta(i)$ of a random-hashed set-associative cache versus the ratio of the fully associative cache size $i$ to the set-associative cache size $n_L$. For example, when we compare random-hashed caches with fully associative caches of the same size, the conflict-miss overhead is around an unimpressive 60% for the curves plotted in Figure 2-4. If we compare random-hashed caches with fully associative caches of half the size, the conflict-miss overhead is around 5% for $S = 4$, and around 40% for $S = 1$. Thus, higher associativity reduces the conflict-miss overhead if we look at proportionally smaller LRU caches. On the other hand, if we compare random-hashed caches with LRU caches bigger by a factor of more than 1.3, caches with smaller associativity exhibit better conflict-miss overheads.

The conflict-miss overhead shown in Figure 2-4 is approximately tight for the worst-case, from Lemma 9 and Corollary 13. These values are not impressive, however, for typical set-associativities between 2 and 4. In the next section, we shall remarkably improve the worst-case conflict-miss overhead using victim caches. In Section 2.6, we shall see that SPEC95 floating point benchmarks perform much better than predicted in Figure 2-4 on random-hashed set-associative caches.

## 2.4 *Victim caches*

In this section, we analyze random-hashed set-associative caches augmented with victim caches [33]. A victim cache is a small fully associative LRU cache that holds items knocked out by the hashed set-associative cache. Let $n_V$ be the number of lines in the victim cache. Intuitively, the advantage of adding a victim cache is to avoid thrashing caused by as many as $n_V$ badly-mapped cache lines, which we refer to as overflowing lines. Victim

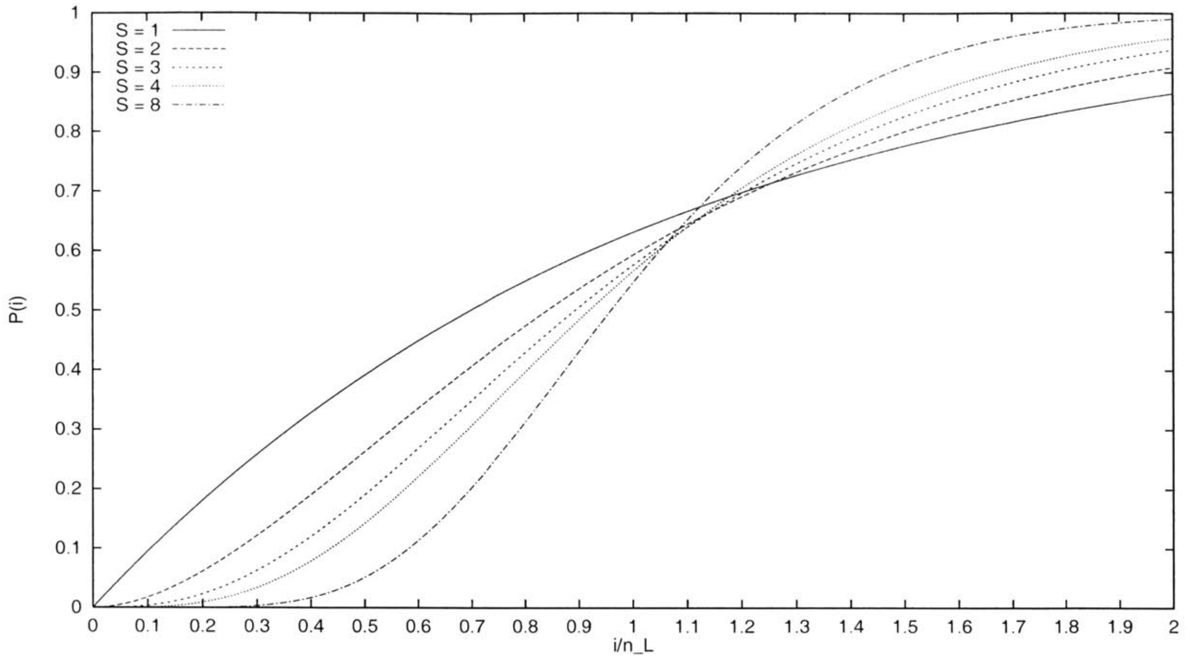



**Figure 2-4:** Plot of $P(i)$ versus $i/n_L$ for set associativity $S = 1, 2, 3, 4, 8$. The random-hashed cache is capable of holding $n_L$ lines. $P(i)$ is also the worst-case conflict-miss overhead $\beta(i)$ of the random-hashed cache with respect to a fully associative cache of size $i$.

cache do not help if all lines in the memory-access sequence are mapped to the same set. Consequently, caches with deterministic line-placement functions perform only as well as a fully associative cache of size $n_V + S$, for certain tailored memory-access sequences. In our cache model, however, victim caches have significant provable gains. Specifically, for $n_V$ greater than a certain threshold, the conflict-miss overhead is reduced by a factor exponential in $n_V$.

The analysis of adding victim caches, presented in this section, works for a large class of probabilistic LRU caches including random-hashed set-associative caches. The probabilistic LRU caches belonging to this class satisfy the following technical condition.

**Definition 15** *A P-LRU cache is said to be* ***negatively correlated on exclusion*** *if during any access, the probability that cache lines with ranks* $i_1, i_2, \ldots, i_k$ *are not present in the cache is at most* $P(i_1)P(i_2)\cdots P(i_k)$.

We now prove that random-hashed set-associative caches satisfy the technical condition given in Definition 15.

**Lemma 16** *Random-hashed set-associative caches are negatively correlated on exclusion.*

*Proof.* Let us consider cache lines with ranks $i_1, i_2, \ldots, i_k$ and estimate the probability that all of them are excluded from the cache. Let event $E_j$ denote the exclusion of the cache line with rank $i_j$.

Consider the probability $\Pr\{E_{j+1}|E_1 \wedge \cdots \wedge E_j\}$. Let line $x$ correspond to event $E_{j+1}$. We are given that a particular set of $j$ cache lines have been thrown out of the cache, even though they have been accessed after $x$. Therefore, if $x$ is mapped to any one of those sets, it would be eliminated even if the $j$ cache lines were not accessed. If $x$ did not map to one of those sets, the $j$ accesses would not matter. Thus, if it is given that cache lines with rank $i_1, i_2, \ldots, i_j$ are excluded, we can ignore the $j$ accesses, whence

$$\begin{aligned}
\Pr\{E_{j+1}|E_1 \wedge \cdots \wedge E_j\} &= P(i_{j+1} - j) \\
&\leq P(i_{j+1}) \\
&= \Pr\{E_{j+1}\} .
\end{aligned}$$

From Bayes' Rule [36, 440], we have

$$\begin{aligned}
\Pr\{E_1 \wedge \cdots \wedge E_k\} &= \prod_{j=0}^{k-1} \Pr\{E_{j+1} \mid E_1 \ldots E_j\} \\
&\leq \prod_{j=0}^{k-1} \Pr\{E_{j+1}\} \\
&\leq P(i_1)P(i_2)\cdots P(i_k) .
\end{aligned}$$

□

We can now bound the number of overflowing lines and prove the worth of adding a victim cache.

**Lemma 17** *Let C be a P-LRU cache negatively correlated on exclusion that is augmented with a victim cache capable of holding $n_V$ lines. Then, C is a $P_V$-LRU cache satisfying*

$$P_V(i) \leq P(i) \cdot \left( \frac{e^{\nu}}{(1+\nu)^{1+\nu}} \right)^{n_V} , \tag{2.3}$$

*where* $1 + \nu = n_V / \sum_{j=0}^{i-1} P(j)$.

*Proof.* Let $x$ be the element accessed in the $t$th memory operation, and let $i = \text{rank}_t(x)$. Let $E_j$ be the event that the cache line with rank $j$ is not present in cache $C$. Let $X_j$ be an

indicator random variable defined as $X_j = Val(E_j)$. Cache $C$ augmented with the victim cache does not hold the cache line with rank $i$ if and only if

1. at least $S$ lines map to $H(x)$, and
2. at least $n_V$ cache lines with rank smaller than $i$ are excluded from the $P$-LRU cache.

Therefore, we have

$$P_V(i) = \Pr\left\{E_i \wedge \left(\sum_{j=0}^{i-1} X_j \geq n_V\right)\right\}$$

which proves that C is a probabilistic LRU cache.

Since events $E_0, E_1, \ldots, E_i$ are negatively correlated, we can upper bound $P_V(a)$ with an analysis where all the events are independent. In particular, we can apply the Chernoff bound on $\sum X_i$ having expectation $\sum P(i)$. The result follows. □

Augmenting a victim cache to a $P$-LRU cache negatively correlated on exclusion reduces $P(i)$ by a factor exponential in the size of the victim cache, provided the size of the victim cache is greater than $\sum_{j=0}^{i-1} P(j)$. We refer to the expression $\sum_{j=0}^{i-1} P(j)$, as the ***expected number of overflowing lines after i accesses***. Lemma 17 applies to all probabilistic LRU caches negatively correlated on exclusion, and consequently it applies to random-hashed set-associative caches (from Lemma 16). A simple upper bound on this summation is $iP(i)$. For small values of $S$, however, we can obtain good approximations by numerically computing the expected number of overflowing lines. Such an approximation, obtained by numerical integration is shown in Figure 2-5. We can see that the function is concave. Figure 2-6 shows the reduction in conflict-miss overhead caused by the addition of victim caches for various values of $\nu$ in (2.3). This graph is very impressive, since the multiplicative factor reduces below a millionth for $\nu = 2$, once the expected number of overflowing lines is greater than 10.

Using the relations shown in Figures 2-4, 2-5, and 2-6, we can compute the absence function $P_V$ for common configurations of caches. An illustrative example is shown below.

**Example 18** Consider a 16KB random-hashed 4-way set-associative cache with 32-byte cache lines. The cache has $n_L = 512$ cache lines. The absence function $P$ associated with the cache is shown in Figure 2-4. Let us augment the cache with an $n_V = 8$ victim cache lines.

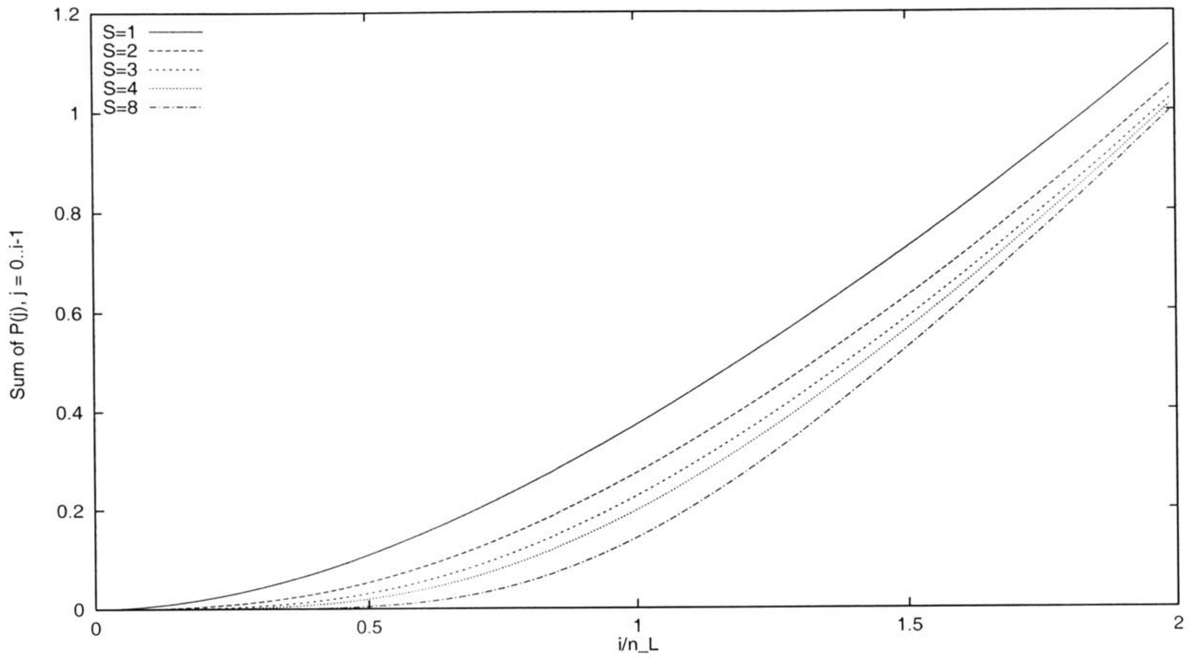



**Figure 2-5:** Graph showing the expected number of overflowing lines after $i$ accesses, divided by $n_L$, versus $i/n_L$.

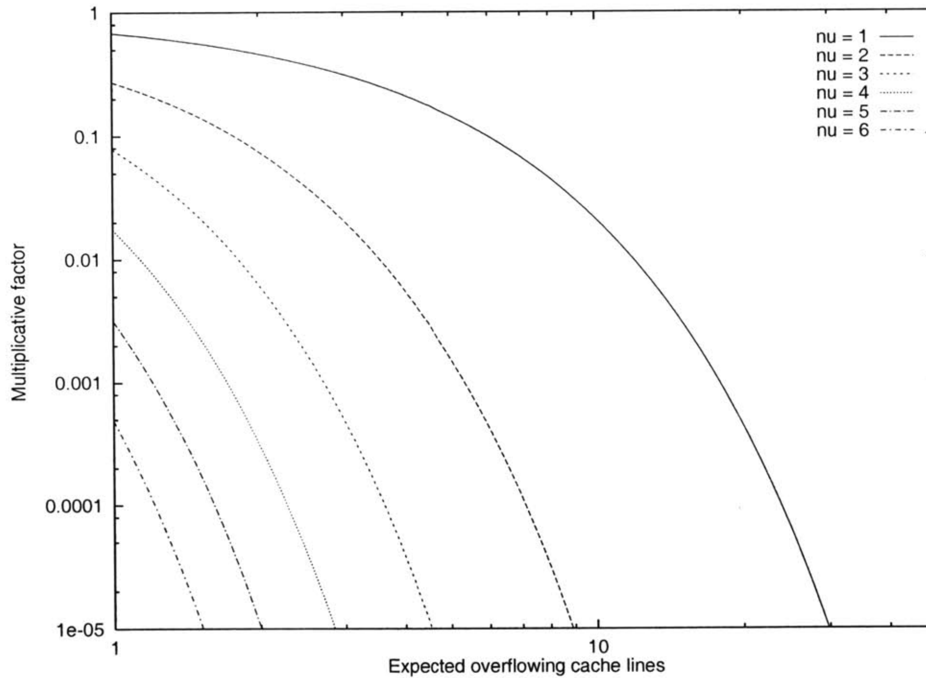



**Figure 2-6:** Multiplicative factor in $P(i)$ caused by the addition of a $(1+\nu)a$-line victim cache, versus the expected number $a$ of overflowing lines after $i$ accesses.

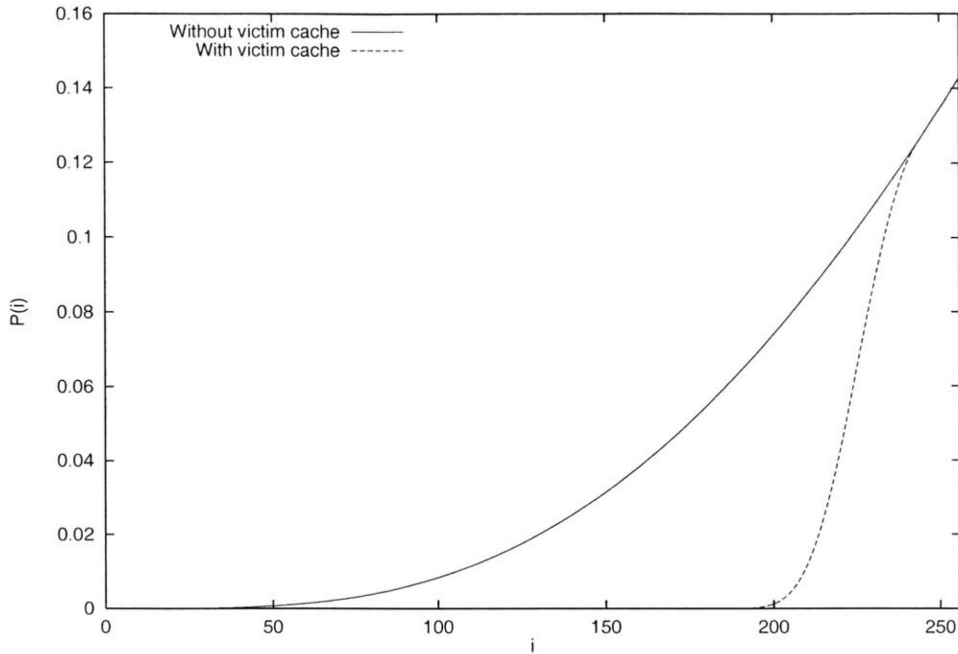



**Figure 2-7:** Graph showing absence function $P(i)$ versus $i$ before and after adding an 8-line victim cache to a 512-line 4-way set-associative random-hashed cache.

The ratio $n_V/n_L \approx 0.16$. Consequently, for all $i$ such that $\sum_{j=0}^{i-1} P(j) < 0.16$, we can apply the victim cache reduction given in (2.3). Figure 2-7 shows how the addition of the victim cache improves the $P(i)$ function. We can see from Figure 2-7 that accesses to items with rank less than 200 are almost certainly cache hits.

## 2.5 *Random-hashed victim caches*

A drawback of fully associative victim caches is their poor scalability. In today's architectures, victim caches hold few cache lines, since they are fully associative. In this section, we shall study victim caches that are not fully associative. Specifically, this section analyzes the performance of random-hashed set-associative victim caches, thereby making a case for set-associative victim caches.

**Lemma 19** *Let C be a P-LRU cache negatively correlated on exclusion, that is augmented with a P′-LRU victim cache. Then, C is a $P_V$-LRU cache satisfying the condition that for any $n_V$, we have*

$$P_V(i) \quad \leq \quad P(i) \cdot \left( \left( \frac{e^{\nu}}{(1+\nu)^{1+\nu}} \right)^{n_V} + P'(n_V) \right) ,$$

*where* $1 + \nu = n_V / \sum_{j=0}^{i-1} P(j)$.

*Proof.* This proof is similar to the proof for Lemma 17. Let $x$ be the element accessed in the $t$th memory operation, and let $i = \text{rank}_t(x)$. Let $E_j$ be the event that the cache line with rank $j$ is not present in cache $C$. Let $X_j$ be an indicator random variable defined by $X_j = Val(E_j)$. Similarly, let us define events $E'_j$ for the victim cache. The cache $C$ augmented with the victim cache does not hold the cache line with rank $i$ if and only if

1. at least $S$ lines map to $H(x)$, and
2. the victim cache does not hold the cache line with rank $j$, where $j$ is the number of cache lines evicted from $C$ with rank smaller than $i$.

Then, we have

$$P_V(i) = \Pr\left\{E_i \wedge E_{\sum_{j=0}^{i-1} X_j}\right\} ,$$

which proves that $C$ is a probabilistic LRU cache. We can upper bound $P_V(i)$ as

$$P_V(i) = \Pr\left\{ E_i \wedge \left( \left( \sum_{j=0}^{i-1} X_j > n_V \right) \vee E_{n_V} \right) \right\} .$$

Since events $E_0, E_1, \ldots, E_i$ are negatively correlated, we can upper bound $P_V(a)$ with an analysis where all the events are independent. In particular, we can apply the Chernoff bound on $\sum X_i$ having expectation $\sum P(i)$, for any suitable $n_V$. □

In (2.4), the parameter $n_V$ is a free variable. In other words, (2.4) works for any value of $n_V$. The choice of an appropriate $n_V$ therefore becomes crucial in obtaining a good upper bound on $P(i)$. In (2.4), the factor attenuating $P(i)$ consists of two terms. The first term decreases with increasing $n_V$, while the second increases with increasing $n_V$. Due to the presence of $n_V$ in the exponent of the first term, we can expect value of $n_V$ to be close to $\sum_{j=0}^{i-1} P(j)$, for large victim caches.

The impact of Lemma 19 is illustrated in the following example.

**Example 20** Consider a 512KB random-hashed 4-way set-associative cache with 32-byte cache lines. The cache has $n_L$ = 16K cache lines. The absence function $P$ associated with the cache is shown in Figure 2-4. Let us augment the cache with an 8-way set-associative random-hashed cache with 256 cache lines. In this example, we choose the optimal value for $n_V$. Figure 2-8 shows the $P(i)$ function before and after adding the victim cache. Notice

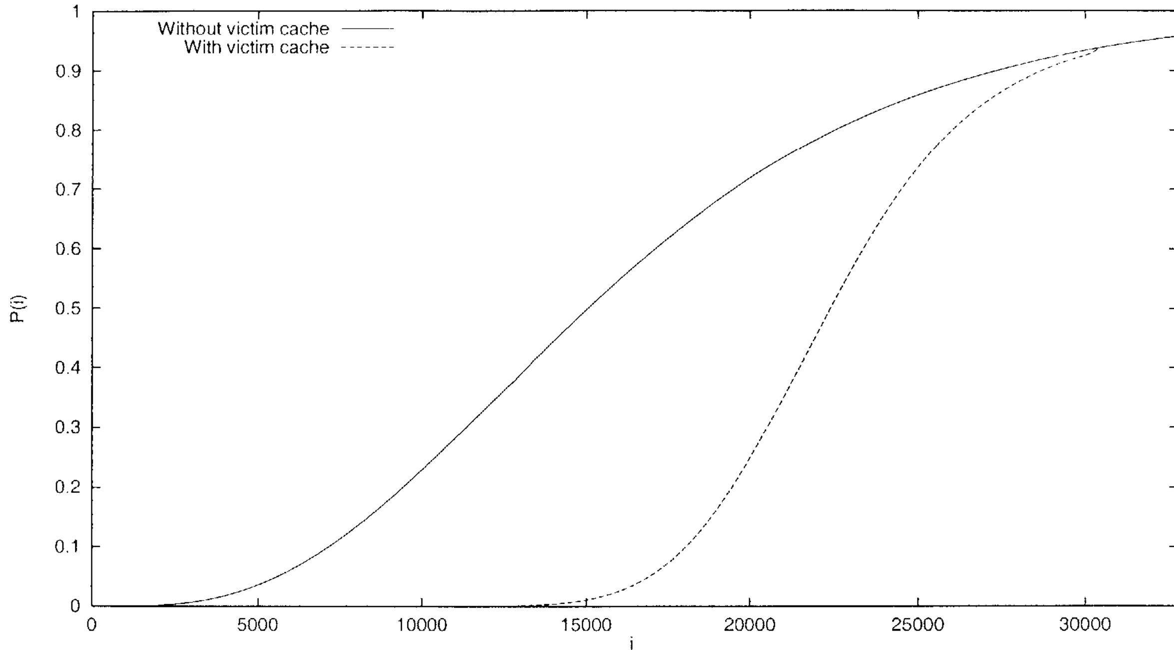



**Figure 2-8:** Graph showing absence function $P(i)$ versus $i$ before and after adding an 256-line 8-way set-associative victim cache to a 16K-line 4-way set-associative random-hashed cache.

that the conflict-miss overhead, when compared to a same size LRU cache, is given by $P(16K)$ which has a very impressive improvement. Moreover, the size of the victim cache is 1/64th the size of the primary cache. Since both caches are set-associative, such an implementation seems possible.

## 2.6 *Simulation results*

Until now, the analysis of random-hashed caches has been independent of the memory-access sequence, providing generic guarantees on performance. Such analysis accounts for the worst-case memory-access sequences (given in Lemma 6), however, and tends to be loose. We can tighten the analysis of expected cache misses using simulation studies for certain memory-access sequences. In this section, we use Lemma 6 to determine the expected number of cache misses for some SPEC95 benchmarks.

The strategy for evaluating good bounds on the expected number of cache misses is as follows. Traces of benchmark programs are simulated to evaluate the cache misses incurred on fully associative LRU caches of all sizes. This data can be plugged in (2.2) to

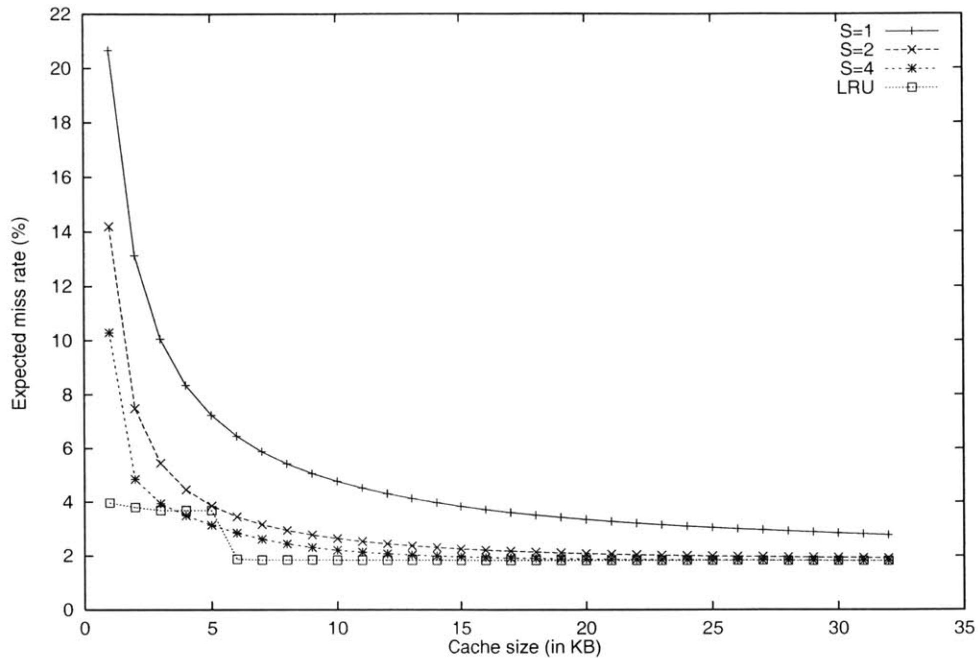



**Figure 2-9:** Graph showing expected miss rate in random-hashed set-associative caches with 32 byte line length, for `mgrid` SPEC floating point benchmark, versus cache size.

give good estimates of the expected number of cache misses. I used the Hierarchical Cache Simulator developed for the Malleable Cache Project [17] at MIT Laboratory for Computer Science. Derek Choiu, a postdoctoral student in the Computer Structures Group, MIT LCS, helped me perform the simulation. The simulator operates on `pdt` traces generated from programs compiled on the `simplescalar` platform [16].

Figure 2-9 shows the expected number of cache misses during 100 million accesses in the `mgrid` SPEC95 floating-point benchmark for various cache sizes and associativity $S = 1, 2$, and 4. The cache line length was assumed to be 32 bytes. With the addition of an 8-line fully associative victim cache, the expected cache misses are as shown in Figure 2-10. Similar results are shown for some other SPEC95 floating point benchmarks in Figures 2-11 through 2-15. For the cache sizes considered, the performance of direct-mapped random-hashed caches with the victim cache is usually better than 4-way set-associative random-hashed caches of same size.

Figure 2-9 and Figure 2-10 exhibit a few anomalies. Firstly, some random-hashed caches outperform fully associative caches for small cache sizes. This phenomenon is caused by the abrupt absence of cache hits having ranks in the range 2K–5K. Random-

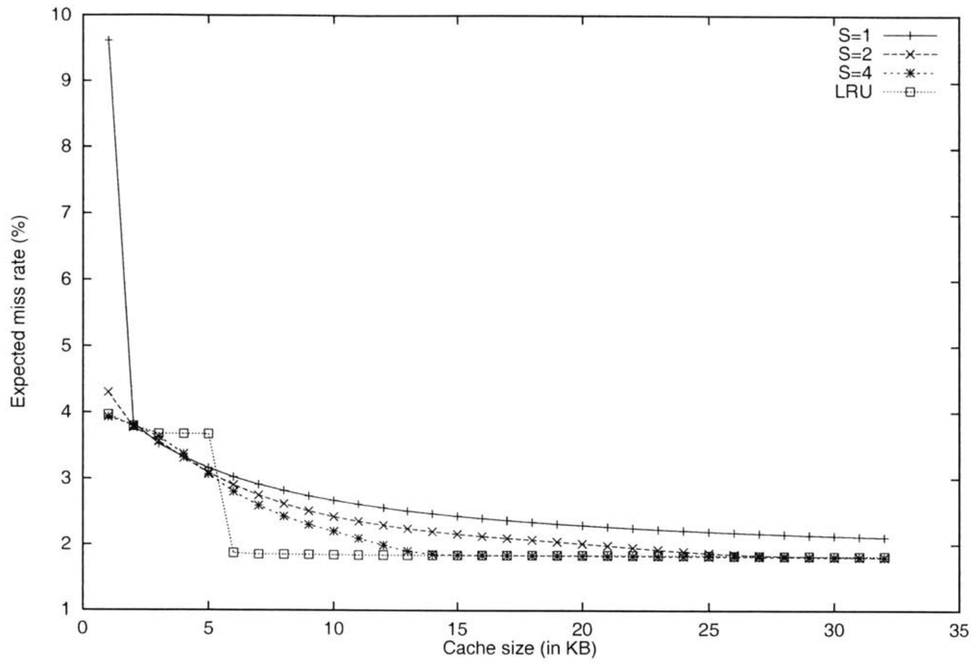



**Figure 2-10:** Graph showing expected miss rate versus cache size for random-hashed set-associative caches with 8-line victim cache and 32-byte line length for `mgrid` SPEC floating point benchmark.

hashed caches that hold around 5K cache lines partially capitalize on cache hits of rank larger than 5K, unlike fully associative caches of the same size. Secondly, for cache sizes between 2K–4K, in Figure 2-10, the expected number of cache misses increases for increasing associativity. Again, this behavior stems from the forbidden rank range 2K–5K. Figure 2-4 shows that smaller associativity caches have smaller absence probability $P(i)$ for $i > n_L$. Since, for $i > n_L$, $P(i)$ weights cache hits with rank larger than 5K in (2.2) caches with lower associativity tend to perform better.

## 2.7 *Conclusion*

In this section, we shall relate the results presented in this chapter to previous and ongoing research on cache design, and we shall look at interesting open problems that deserve scrutiny. Specifically, we shall compare our results with the analytical cache model proposed by Agarwal, Horowitz, and Hennessy [2]. The independence of our analysis from the memory-access sequence is both a strength and weakness. We shall discuss this issue. We shall also look at other caching mechanisms that achieve similar performance, thereby

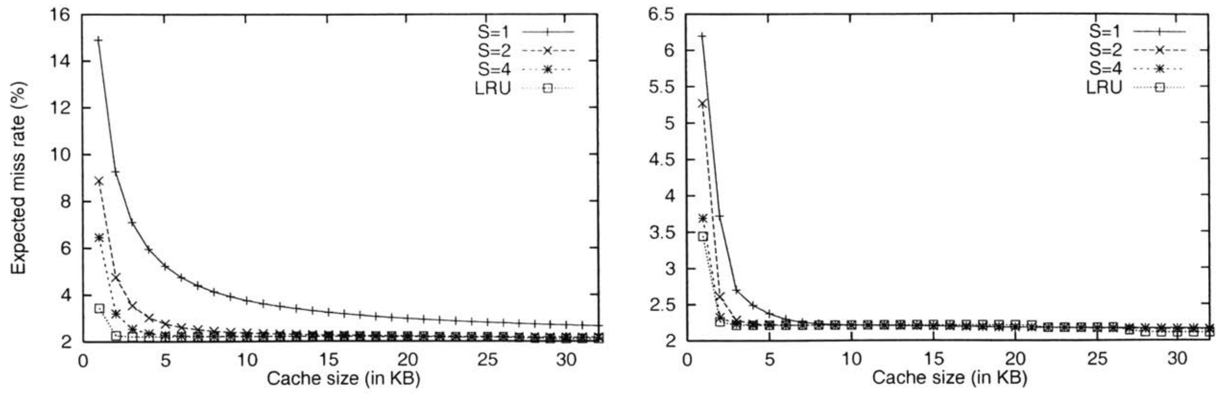



**Figure 2-11:** Expected miss rate incurred by `applu` versus cache size. Right hand graph includes an 8-line victim cache.

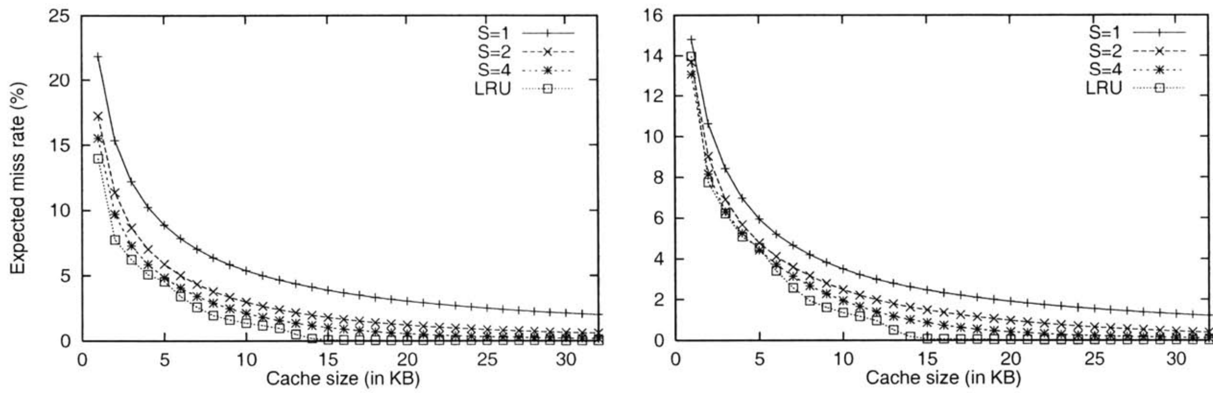



**Figure 2-12:** Expected miss rate incurred by `fpppp` versus cache size. Right hand graph includes an 8-line victim cache.

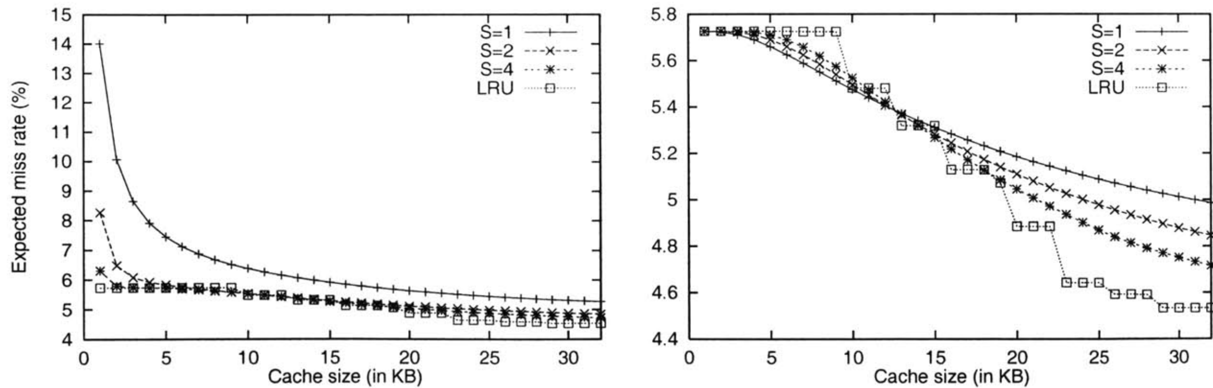



**Figure 2-13:** Expected miss rate incurred by `hydro2d` versus cache size. Right hand graph includes an 8-line victim cache.

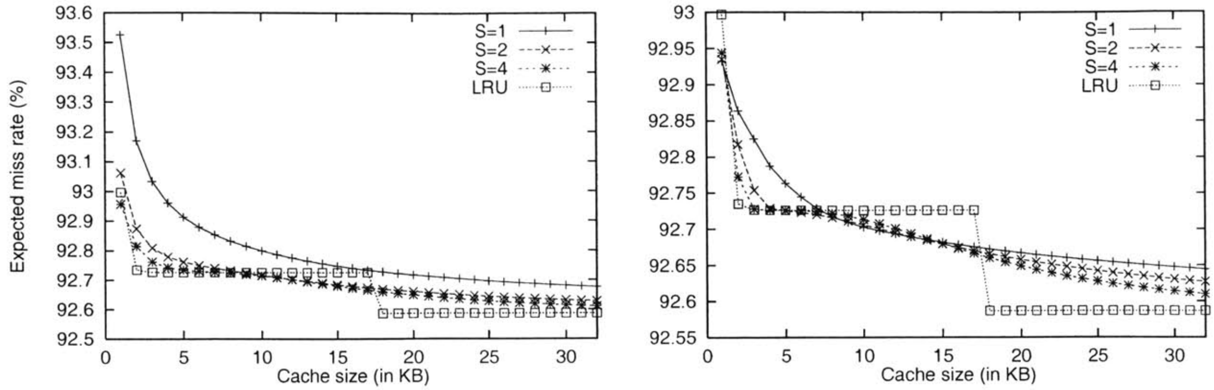



**Figure 2-14:** Expected miss rate incurred by `su2cor` versus cache size. Right hand graph includes an 8-line victim cache.

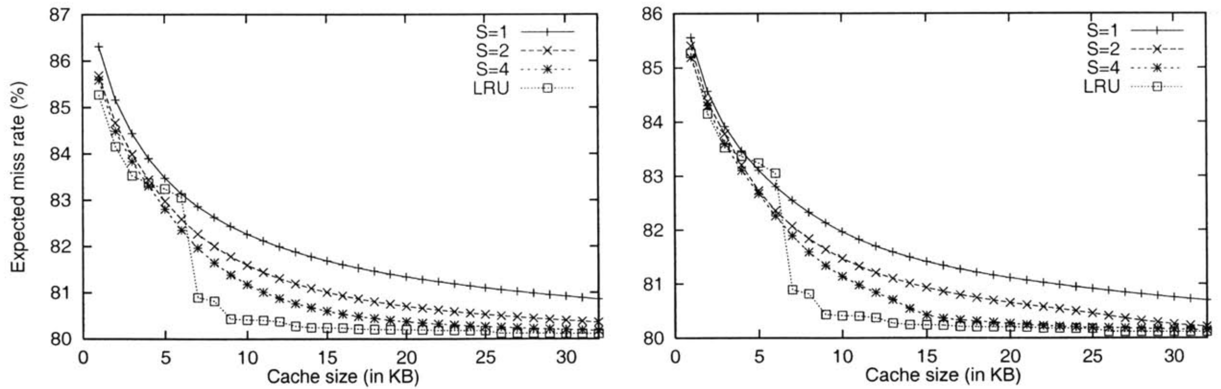



**Figure 2-15:** Expected miss rate incurred by `tomcatv` versus cache size. Right hand graph includes an 8-line victim cache.

citing the possibility of further improvement.

Random-hashed set-associative victim caches, presented in Section 2.5, is an interesting offshoot of my research in random-hashed caches. I believe that set-associative victim caches with pseudorandom line-placement functions may be useful for L2 and L3 caches, which are typically large. I can foresee two practical difficulties, however. Firstly, the gains in introducing such victim caches may be outweighed by the loss of valuable time in moving data between the primary cache and its victim cache. Secondly, victim caches added to ensure atomicity in memory-addressing instructions may not benefit from this result.

Theoretical results related to random-hashed caches are available in the literature. In their analytical cache model, Agarwal, Horowitz, and Hennessy [2] study the collision probability, which is same as the expected conflict-miss overhead, in random-hashed caches. Their analysis, however, makes assumptions regarding the memory-access sequence presented to the cache, such as the size of the working set. The weakness of this analysis lies in the assumptions made on the memory-access sequence.

The analysis of conflict-miss overhead presented in this chapter is independent of the memory-access sequence and can therefore be termed as competitive. To my best knowledge, the results presented in this chapter are the first comprehensive competitiveness study of caches with limited associativity with respect to fully associative caches. A drawback in competitive analysis is that the worst-case performance is used as the metric. Recall that Lemma 9 and Corollary 13 show that our analysis of set-associative random-hashed caches is tight for the worst-case, although Figures 2-9 through 2-15 exhibit much smaller worst-case conflict-miss overhead shown in Figure 2-4. The analysis of fully associative LRU suffers from the same drawback. Sleator and Tarjan's competitive analysis [45] is loose for the average case [54].

Another drawback of our analysis is that conflict-miss overhead is defined with respect to fully associative LRU caches. With this definition, conflict-miss overhead can be negative. For certain cache sizes, random-hash caches outperform fully associative LRU caches of the same size for many SPEC95 floating point benchmarks including `mgrid`, `hydro2d`, `su2cor`, and `tomcatv`. A arguably better definition of conflict-miss overhead will be with respect to ideal caches. An interesting open problem is to obtain competitive analysis that is better than the combination of the analysis of random-hashed caches with respect to

LRU caches, and that of LRU caches with respect to ideal caches. Researchers working on the Malleable Cache Project [17] believe that caching mechanisms better than LRU can be designed if the algorithm provides appropriate directives to the cache. This opens up the possibility of better theoretical analysis of caching mechanisms based on the assumption that accurate directives are provided to the cache.

Finally, our analyses assume that the line-placement function $H$ is a perfectly random hash function. Typically only pseudorandom hash functions can be implemented in practice. I believe, however, that pseudorandom hash functions will exhibit expected performance close to the theoretical predictions. Sacrificing "random behavior" of the hash function for simplicity in implementation is an interesting tradeoff. Such a tradeoff is presented in [27], where the authors compare line-placement functions based on XOR with more complicated ones based on polynomial arithmetic [42].

I believe that set-associative victim caches with pseudorandom line-placement functions may be useful for L2 and L3 caches, which are typically large. I can foresee two practical difficulties, however. Firstly, the gains in introducing such victim caches may be outweighed by the loss of valuable time in moving data between the primary cache and its victim cache. Secondly, victim caches added to ensure atomicity in memory-addressing instructions may not benefit from this result.

Section 4.3 presents an alternative method of simulating LRU caches on direct memory. This method only requires 2-universal hash functions and does not incur any conflict miss overhead when compared to proportionally small LRU caches. I believe, however, that this caching mechanism does not have practical hardware implementations, although it hints the possibility of other caching mechanisms superior to random-hashed caches.

CHAPTER 3

# *Cache-oblivious algorithms*

In the previous chapter, we have looked at how caches with performance close to the ideal-cache model can be designed, justifying the use of ideal-cache model as an abstraction of caches. This chapter presents efficient, portable algorithms for matrix multiplication, matrix transposition, FFT, and sorting. These algorithms are shown to be "optimal" for the ideal-cache model using theoretical analysis, justifying the usefulness of the ideal-cache model as a tool for algorithm design.

Resource-oblivious algorithms that nevertheless use resources efficiently offer advantages of simplicity and portability over resource-aware algorithms whose resource usage must be programmed explicitly. We define an algorithm to be ***cache aware*** if it contains parameters (set at either compile-time or runtime) that can be tuned to optimize the cache complexity for the particular cache size and line length. Otherwise, the algorithm is ***cache oblivious***. Historically, good performance has been obtained using cache-aware algorithms, but we shall exhibit several optimal[1] cache-oblivious algorithms.

To illustrate the notion of cache awareness, consider the problem of multiplying two $n \times n$ matrices $A$ and $B$ to produce their $n \times n$ product $C$. We assume that the three matrices are stored in row-major order. We further assume that $n$ is "big," i.e., $n > L$, in order to simplify the analysis. The conventional way to multiply matrices on a computer with

[1] For simplicity, we use the term "optimal" as a synonym for "asymptotically optimal," since all our analyses are asymptotic.

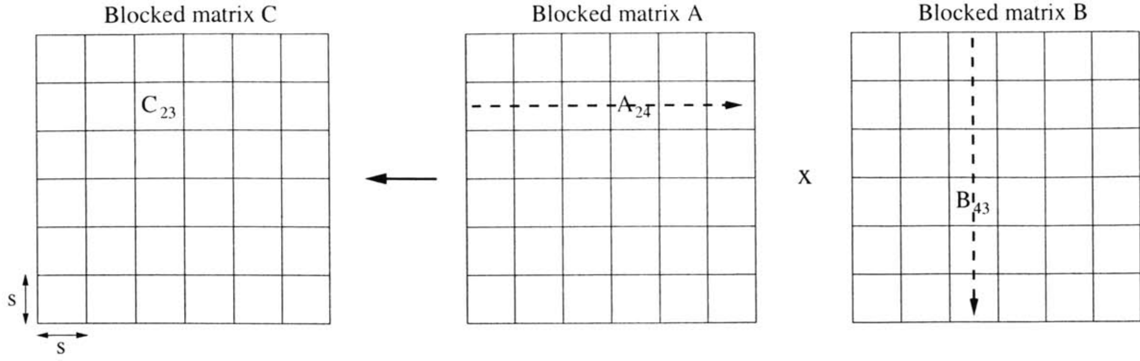



**Figure 3-1:** Blocked matrix multiplication: Iteration when $i = 2$, $j = 3$ and $k = 4$.

caches is to use a ***blocked*** algorithm [26, p. 45]. The idea is to view each matrix $M$ as consisting of $(n/s) \times (n/s)$ submatrices $M_{ij}$ (the blocks), each of which has size $s \times s$, where $s$ is a tuning parameter. The following algorithm implements this strategy:

BLOCK-MULT$(A, B, C, n)$
1 **for** $i \leftarrow 1$ **to** $n/s$
2 **do for** $j \leftarrow 1$ **to** $n/s$
3 **do for** $k \leftarrow 1$ **to** $n/s$
4 **do** ORD-MULT$(A_{ik}, B_{kj}, C_{ij}, s)$

The ORD-MULT$(A, B, C, s)$ subroutine computes $C \leftarrow C + AB$ on $s \times s$ matrices using the ordinary $O(s^3)$ algorithm. This algorithm assumes for simplicity that $s$ evenly divides $n$, but in practice $s$ and $n$ need have no special relationship, yielding more complicated code in the same spirit. A step of this algorithm is shown in Figure 3-1.

Depending on the cache size of the machine on which BLOCK-MULT is run, the parameter $s$ can be tuned to make the algorithm run fast, and thus BLOCK-MULT is a cache-aware algorithm. To minimize the cache complexity, we choose $s$ to be the largest value such that the three $s \times s$ submatrices simultaneously fit in cache. An $s \times s$ submatrix is stored on $O(s + s^2/L)$ cache lines. From the tall-cache assumption (1.1), we can see that $s = \Omega(\sqrt{Z})$. Thus, each of the calls to ORD-MULT runs with at most $Z/L = O(1 + s^2/L)$ cache misses needed to bring the three matrices into the cache. Consequently, the cache complexity of the entire algorithm is $O(1 + n^2/L + (n/\sqrt{Z})^3(Z/L)) = O(1 + n^2/L + n^3/L\sqrt{Z})$, since the algorithm has to read $n^2$ elements, which reside on $\lceil n^2/L \rceil$ cache lines.

The same bound can be achieved using a simple recursive cache-oblivious algorithm

that requires no voodoo tuning parameters such as the $s$ in BLOCK-MULT. Consider the following recursive algorithm for matrix multiplication.

REC-MULT($A, B, C, n$)
1 **if** $n = 1$
2 **then** $C \leftarrow C + AB$
3 **else** $A \equiv \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B \equiv \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C \equiv \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$
4 **for** $i, j, k \in \{1, 2\}$
5 **do** REC-MULT($A_{ik}, B_{kj}, C_{ij}, n/2$)

We shall now analyze the above algorithm in the ideal-cache model. Since the cache is managed by optimal replacement, if all the three matrices fit in cache, they will be loaded into the cache only once. Therefore, if $n$ is smaller than $s$, the matrix multiply incurs only $O(1 + n^2/L)$ cache misses. Notice that the optimal block size $s$ is used only in the analysis and does not appear in the algorithm. If $n$ is larger than $s$, there is a constant number of cache misses apart from the cache misses made by the recursive calls. Thus, the number of cache misses is given by the recurrence

$$Q(n) = \begin{cases} O(1 + n^2/L) & \text{if } n \leq s , \\ 8Q(n/2) + O(1) & \text{otherwise} ; \end{cases}$$

implying that $Q(n) = O(1 + n^2/L + n^3/(sL))$, which matches the bound achieved by the cache-aware algorithm. Chapter 3 gives an efficient cache-oblivious algorithm rectangular matrix multiplication.

The rest of this chapter is as follows. Section 3.1 presents a simple algorithm for general rectangular matrix multiplication. The problems of computing a matrix transpose and of performing an FFT also succumb to remarkably simple cache-oblivious algorithms, which are described in Section 3.2. Sections 3.3 and 3.4 present two sorting algorithms, one based on mergesort and the other on distribution sort, both of which are optimal in both work and cache misses. The results presented in this section represent joint work with Frigo, Leiserson, and Prokop. [25].

## 3.1 Matrix multiplication

This section describes and analyzes a cache-oblivious algorithm for multiplying an $m \times n$ matrix by an $n \times p$ matrix cache-obliviously using $\Theta(mnp)$ work and incurring $\Theta(m + n + p + (mn + np + mp)/L + mnp/L\sqrt{Z})$ cache misses. These results require the tall-cache assumption (1.1) for matrices stored in row-major layout format, but the assumption can be relaxed for certain other layouts. I shall also show that Strassen's algorithm [47] for multiplying $n \times n$ matrices, which uses $\Theta(n^{\lg 7})$ work, incurs $\Theta(1 + n^2/L + n^{\lg 7}/L\sqrt{Z})$ cache misses.

In [12], optimal divide-and-conquer algorithm for $n \times n$ matrix multiplication, containing no tuning parameters, was analyzed, but the authors did not study cache-obliviousness *per se*. That algorithm can be extended to multiply rectangular matrices. To multiply a $m \times n$ matrix $A$ and a $n \times p$ matrix $B$, the REC-MULT algorithm halves the largest of the three dimensions and recurs according to one of the following three cases:

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} , \tag{3.1}$$

$$\begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2 , \tag{3.2}$$

$$A \begin{pmatrix} B_1 & B_2 \end{pmatrix} = \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} . \tag{3.3}$$

In case (3.1), we have $m \geq \max\{n, p\}$. Matrix $A$ is split horizontally, and both halves are multiplied by matrix $B$. In case (3.2), we have $n \geq \max\{m, p\}$. Both matrices are split, and the two halves are multiplied. In case (3.3), we have $p \geq \max\{m, n\}$. Matrix $B$ is split vertically, and each half is multiplied by $A$. For square matrices, these three cases together are equivalent to the recursive multiplication algorithm described in [12]. The base case occurs when $m = n = p = 1$, in which case the two elements are multiplied and added into the result matrix.

Although this straightforward divide-and-conquer algorithm contains no tuning parameters, it uses cache optimally. To analyze the REC-MULT algorithm, we assume that the three matrices are stored in row-major order. Intuitively, REC-MULT uses the cache effectively, because once a subproblem fits into the cache, its smaller subproblems can be solved in cache with no further cache misses.

**Theorem 21** *The* REC-MULT *algorithm uses* $\Theta(mnp)$ *work and incurs* $\Theta(m+n+p+(mn+np+mp)/L+mnp/L\sqrt{Z})$ *cache misses when multiplying an* $m \times n$ *matrix by an* $n \times p$ *matrix.*

*Proof.* It can be shown by induction that the work of REC-MULT is $\Theta(mnp)$. To analyze the cache misses, let $\alpha > 0$ be the largest constant sufficiently small that three submatrices of sizes $m' \times n'$, $n' \times p'$, and $m' \times p'$, where $\max\{m', n', p'\} \leq \alpha\sqrt{Z}$, all fit completely in the cache. Four cases arise based on the initial size of the matrices.

***Case I:*** $m, n, p > \alpha\sqrt{Z}$. This case is the most intuitive. The matrices do not fit in cache, since all dimensions are "big enough." The cache complexity can be described by the recurrence

$$Q(m,n,p) \leq \begin{cases} \Theta((mn+np+mp)/L) & \text{if } m,n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m/2,n,p)+O(1) & \text{ow. if } m \geq n \text{ and } m \geq p \ , \\ 2Q(m,n/2,p)+O(1) & \text{ow. if } n > m \text{ and } n \geq p \ , \\ 2Q(m,n,p/2)+O(1) & \text{otherwise} \ . \end{cases}$$

The base case arises as soon as all three submatrices fit in cache. The total number of lines used by the three submatrices is $\Theta((mn+np+mp)/L)$. The only cache misses that occur during the remainder of the recursion are the $\Theta((mn+np+mp)/L)$ cache misses required to bring the matrices into cache. In the recursive cases, when the matrices do not fit in cache, we pay for the cache misses of the recursive calls, which depend on the dimensions of the matrices, plus $O(1)$ cache misses for the overhead of manipulating submatrices. The solution to this recurrence is $Q(m,n,p) = \Theta(mnp/L\sqrt{Z})$.

***Case II:*** ($m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z}$) or ($n \leq \alpha\sqrt{Z}$ and $m, p > \alpha\sqrt{Z}$) or ($p \leq \alpha\sqrt{Z}$ and $m, n > \alpha\sqrt{Z}$). Here, we consider the case where $m \leq \alpha\sqrt{Z}$ and $n, p > \alpha\sqrt{Z}$. The proofs for the other cases are only small variations of this proof. The REC-MULT algorithm always divides $n$ or $p$ by 2 according to cases (3.2) and (3.3). At some point in the recursion, both are small enough that the whole problem fits into cache. The number of cache misses can be described by the recurrence

$$Q(m,n,p) \leq \begin{cases} \Theta(1+n+np/L+m) & \text{if } n,p \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] \ , \\ 2Q(m,n/2,p)+O(1) & \text{otherwise if } n \geq p \ , \\ 2Q(m,n,p/2)+O(1) & \text{otherwise} \ ; \end{cases}$$

whose solution is $Q(m,n,p) = \Theta(np/L + mnp/L\sqrt{Z})$.

***Case III:*** $(n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z})$ or $(m, p \leq \alpha\sqrt{Z}$ and $n > \alpha\sqrt{Z})$ or $(m, n \leq \alpha\sqrt{Z}$ and $p > \alpha\sqrt{Z})$. In each of these cases, one of the matrices fits into cache, and the others do not. Here, we consider the case where $n, p \leq \alpha\sqrt{Z}$ and $m > \alpha\sqrt{Z}$. The other cases can be proven similarly. The REC-MULT algorithm always divides $m$ by 2 according to case (3.1). At some point in the recursion, $m$ falls into the range $\alpha\sqrt{Z}/2 \leq m \leq \alpha\sqrt{Z}$, and the whole problem fits in cache. The number cache misses can be described by the recurrence

$$Q(m,n) \leq \begin{cases} \Theta(1+m) & \text{if } m \in [\alpha\sqrt{Z}/2, \alpha\sqrt{Z}] , \\ 2Q(m/2,n,p) + O(1) & \text{otherwise ;} \end{cases}$$

whose solution is $Q(m,n,p) = \Theta(m + mnp/L\sqrt{Z})$.

***Case IV:*** $m, n, p \leq \alpha\sqrt{Z}$. From the choice of $\alpha$, all three matrices fit into cache. The matrices are stored on $\Theta(1 + mn/L + np/L + mp/L)$ cache lines. Therefore, we have $Q(m,n,p) = \Theta(1 + (mn + np + mp)/L)$. □

We require the tall-cache assumption (1.1) in these analyses, because the matrices are stored in row-major order. Tall caches are also needed if matrices are stored in column-major order , but the assumption that $Z = \Omega(L^2)$ can be relaxed for certain other matrix layouts. The $s \times s$-blocked layout , for some tuning parameter $s$, can be used to achieve the same bounds with the weaker assumption that the cache holds at least some sufficiently large constant number of lines. The cache-oblivious bit-interleaved layout has the same advantage as the blocked layout, but no tuning parameter need be set, since submatrices of size $O(\sqrt{L}) \times O(\sqrt{L})$ are cache-obliviously stored on $O(1)$ cache lines. The advantages of bit-interleaved and related layouts have been studied in [13, 14, 22]. One of the practical disadvantages of bit-interleaved layouts is that index calculations on conventional microprocessors can be costly, a deficiency we hope that processor architects will remedy.

For square matrices, the cache complexity $Q(n) = \Theta(n + n^2/L + n^3/L\sqrt{Z})$ of the REC-MULT algorithm is the same as the cache complexity of the cache-aware BLOCK-MULT algorithm and also matches the lower bound by Hong and Kung [31]. This lower bound holds for all algorithms that execute the $\Theta(n^3)$ operations given by the definition of matrix multiplication

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \; .$$

No tight lower bounds for the general problem of matrix multiplication are known.

By using an asymptotically faster algorithm, such as Strassen's algorithm [47] or one of its variants [53], both the work and cache complexity can be reduced. When multiplying $n \times n$ matrices, Strassen's algorithm, which is cache oblivious, requires only 7 recursive multiplications of $n/2 \times n/2$ matrices and a constant number of matrix additions, yielding the recurrence

$$Q(n) \le \begin{cases} \Theta(1 + n + n^2/L) & \text{if } n^2 \le \alpha Z \,, \\ 7Q(n/2) + O(n^2/L) & \text{otherwise} \,; \end{cases} \tag{3.4}$$

where $\alpha$ is a sufficiently small constant. The solution to this recurrence is $\Theta(n + n^2/L + n^{\lg 7}/L\sqrt{Z})$.

## 3.2 *Matrix transposition and FFT*

This section describes a recursive cache-oblivious algorithm for transposing an $m \times n$ matrix which uses $O(mn)$ work and incurs $O(1 + mn/L)$ cache misses, which is optimal. Using matrix transposition as a subroutine, we convert a variant [52] of the "six-step" fast Fourier transform (FFT) algorithm [9] into an optimal cache-oblivious algorithm. This FFT algorithm uses $O(n \lg n)$ work and incurs

$$O\left(1 + (n/L)\left(1 + \log_Z n\right)\right)$$

cache misses.

The problem of matrix transposition is defined as follows. Given an $m \times n$ matrix stored in a row-major layout, compute and store $A^T$ into an $n \times m$ matrix $B$ also stored in a row-major layout. The straightforward algorithm for transposition that employs doubly nested loops incurs $\Theta(mn)$ cache misses on one of the matrices when $m \gg Z/L$ and $n \gg Z/L$, which is suboptimal.

Optimal work and cache complexities can be obtained with a divide-and-conquer strategy, however. If $n \ge m$, the REC-TRANSPOSE algorithm partitions

$$A = (A_1 \; A_2) \,, \quad B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

and recursively executes REC-TRANSPOSE$(A_1, B_1)$ and REC-TRANSPOSE$(A_2, B_2)$. Otherwise, it divides matrix $A$ horizontally and matrix $B$ vertically and likewise performs two

transpositions recursively. The next two lemmas provide upper and lower bounds on the performance of this algorithm.

**Lemma 22** *The* REC-TRANSPOSE *algorithm involves $O(mn)$ work and incurs $O(1+mn/L)$ cache misses for an $m \times n$ matrix.*

*Proof.* That the algorithm does $O(mn)$ work is straightforward. For the cache analysis, let $Q(m,n)$ be the cache complexity of transposing an $m \times n$ matrix. We assume that the matrices are stored in row-major order, the column-major layout having a similar analysis.

Let $\alpha$ be a constant sufficiently small such that two submatrices of size $m \times n$ and $n \times m$, where $\max\{m,n\} \leq \alpha L$, fit completely in the cache even if each row is stored in a different cache line. We distinguish the three cases.

***Case I:*** $\max\{m,n\} \leq \alpha L$. Both the matrices fit in $O(1)+2mn/L$ lines. From the choice of $\alpha$, the number of lines required is at most $Z/L$. Therefore $Q(m,n) = O(1+mn/L)$.

***Case II:*** $m \leq \alpha L < n$ or $n \leq \alpha L < m$. Suppose first that $m \leq \alpha L < n$. The REC-TRANSPOSE algorithm divides the greater dimension $n$ by 2 and performs divide and conquer. At some point in the recursion, $n$ falls into the range $\alpha L/2 \leq n \leq \alpha L$, and the whole problem fits in cache. Because the layout is row-major, at this point the input array has $n$ rows and $m$ columns, and it is laid out in contiguous locations, requiring at most $O(1+nm/L)$ cache misses to be read. The output array consists of $nm$ elements in $m$ rows, where in the worst case every row lies on a different cache line. Consequently, we incur at most $O(m+nm/L)$ for writing the output array. Since $n \geq \alpha L/2$, the total cache complexity for this base case is $O(1+m)$. These observations yield the recurrence

$$Q(m,n) \leq \begin{cases} O(1+m) & \text{if } n \in [\alpha L/2, \alpha L] \, , \\ 2Q(m,n/2)+O(1) & \text{otherwise} \, ; \end{cases}$$

whose solution is $Q(m,n) = O(1+mn/L)$.

The case $n \leq \alpha L < m$ is analogous.

***Case III:*** $m,n > \alpha L$. As in Case II, at some point in the recursion both $n$ and $m$ fall into the range $[\alpha L/2, \alpha L]$. The whole problem fits into cache and can be solved with at most

$O(m+n+mn/L)$ cache misses. The cache complexity thus satisfies the recurrence

$$Q(m,n) \leq \begin{cases} O(m+n+mn/L) & \text{if } m,n \in [\alpha L/2, \alpha L] , \\ 2Q(m/2,n)+O(1) & \text{if } m \geq n , \\ 2Q(m,n/2)+O(1) & \text{otherwise;} \end{cases}$$

whose solution is $Q(m,n) = O(1+mn/L)$. □

**Theorem 23** *The* REC-TRANSPOSE *algorithm has optimal cache complexity.*

*Proof.* For an $m \times n$ matrix, the algorithm must write to $mn$ distinct elements, which occupy at least $\lceil mn/L \rceil = \Omega(1+mn/L)$ cache lines. □

As an example of an application of this cache-oblivious transposition algorithm, in the rest of this section we look at a cache-oblivious algorithm for computing the discrete Fourier transform of a complex array of $n$ elements, where $n$ is an exact power of 2. The basic algorithm is the well-known "six-step" variant [9, 52] of the Cooley-Tukey FFT algorithm [18]. Using the cache-oblivious transposition algorithm, however, the FFT becomes cache-oblivious, and its performance matches the lower bound by Hong and Kung [31].

Recall that the ***discrete Fourier transform (DFT)*** of an array $X$ of $n$ complex numbers is the array $Y$ given by

$$Y[i] = \sum_{j=0}^{n-1} X[j]\omega_n^{-ij} , \tag{3.5}$$

where $\omega_n = e^{2\pi\sqrt{-1}/n}$ is a primitive $n$th root of unity, and $0 \leq i < n$. Many algorithms evaluate Equation (3.5) in $O(n \lg n)$ time for all integers $n$ [20]. We assume that $n$ is an exact power of 2, however, and compute Equation (3.5) according to the Cooley-Tukey algorithm, which works recursively as follows. In the base case where $n = O(1)$, we compute Equation (3.5) directly. Otherwise, for any factorization $n = n_1 n_2$ of $n$, we have

$$Y[i_1 + i_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[ \left( \sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2]\omega_{n_1}^{-i_1 j_1} \right) \omega_n^{-i_1 j_2} \right] \omega_{n_2}^{-i_2 j_2} . \tag{3.6}$$

Observe that both the inner and outer summations in Equation (3.6) are DFT's. Operationally, the computation specified by Equation (3.6) can be performed by computing $n_2$ transforms of size $n_1$ (the inner sum), multiplying the result by the factors $\omega_n^{-i_1 j_2}$ (called the ***twiddle factors*** [20]), and finally computing $n_1$ transforms of size $n_2$ (the outer sum).

We choose $n_1$ to be $2^{\lceil \lg n/2 \rceil}$ and $n_2$ to be $2^{\lfloor \lg n/2 \rfloor}$. The recursive step then operates as follows:

1. Pretend that input is a row-major $n_1 \times n_2$ matrix $A$. Transpose $A$ in place, i.e., use the cache-oblivious REC-TRANSPOSE algorithm to transpose $A$ onto an auxiliary array $B$, and copy $B$ back onto $A$. Notice that if $n_1 = 2n_2$, we can consider the matrix to be made up of records containing two elements.

2. At this stage, the inner sum corresponds to a DFT of the $n_2$ rows of the transposed matrix. Compute these $n_2$ DFT's of size $n_1$ recursively. Observe that, because of the previous transposition, we are transforming a contiguous array of elements.

3. Multiply $A$ by the twiddle factors, which can be computed on the fly with no extra cache misses.

4. Transpose $A$ in place, so that the inputs to the next stage are arranged in contiguous locations.

5. Compute $n_1$ DFT's of the rows of the matrix recursively.

6. Transpose $A$ in place so as to produce the correct output order.

It can be proven by induction that the work complexity of this FFT algorithm is $O(n \lg n)$. We now analyze its cache complexity. The algorithm always operates on contiguous data, by construction. Thus, by the tall-cache assumption (1.1), the transposition operations and the twiddle-factor multiplication require at most $O(1+n/L)$ cache misses. Thus, the cache complexity satisfies the recurrence

$$Q(n) \leq \begin{cases} O(1+n/L), & \text{if } n \leq \alpha Z \text{ ,} \\ n_1 Q(n_2) + n_2 Q(n_1) + O(1+n/L) & \text{otherwise ;} \end{cases} \tag{3.7}$$

where $\alpha > 0$ is a constant sufficiently small that a subproblem of size $\alpha Z$ fits in cache. This recurrence has solution

$$Q(n) = O\big(1 + (n/L)\big(1 + \log_Z n\big)\big) \text{ ,}$$

which is optimal for a Cooley-Tukey algorithm, matching the lower bound by Hong and Kung [31] when $n$ is an exact power of 2. As with matrix multiplication, no tight lower bounds for cache complexity are known for the general DFT problem.

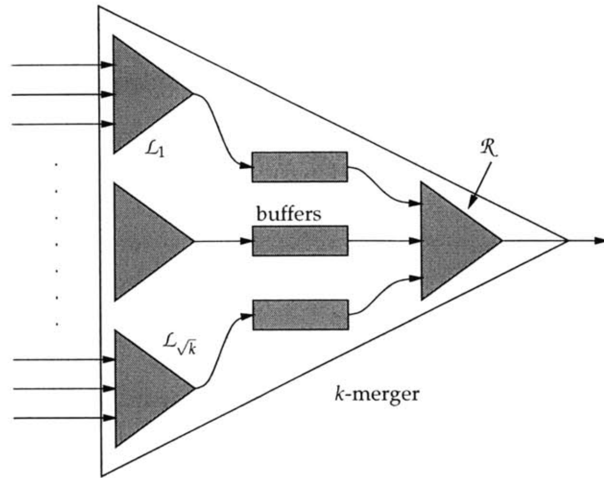



**Figure 3-2:** Illustration of a $k$-merger. A $k$-merger is built recursively out of $\sqrt{k}$ "left" $\sqrt{k}$-mergers $L_1, L_2, \ldots, L_{\sqrt{k}}$, a series of buffers, and one "right" $\sqrt{k}$-merger $R$.

## 3.3 *Funnelsort*

Cache-oblivious algorithms, like the familiar two-way merge sort, are not optimal with respect to cache misses. The $Z$-way mergesort suggested by Aggarwal and Vitter [6] has optimal cache complexity, but although it apparently works well in practice [35], it is cache aware. This section describes a cache-oblivious sorting algorithm called "funnelsort." This algorithm has optimal $O(n \lg n)$ work complexity, and optimal $O(1 + (n/L)(1 + \log_Z n))$ cache complexity.

Funnelsort is similar to mergesort. In order to sort a (contiguous) array of $n$ elements, funnelsort performs the following two steps:

1. Split the input into $n^{1/3}$ contiguous arrays of size $n^{2/3}$, and sort these arrays recursively.

2. Merge the $n^{1/3}$ sorted sequences using a $n^{1/3}$-merger, which is described below.

Funnelsort differs from mergesort in the way the merge operation works. Merging is performed by a device called a ***k*-*merger***, which inputs $k$ sorted sequences and merges them. A $k$-merger operates by recursively merging sorted sequences which become progressively longer as the algorithm proceeds. Unlike mergesort, however, a $k$-merger suspends work on a merging subproblem when the merged output sequence becomes "long enough" and resumes work on another merging subproblem.

This complicated flow of control makes a $k$-merger a bit tricky to describe. Figure 3-2 shows a representation of a $k$-merger, which has $k$ sorted sequences as inputs. Throughout its execution, the $k$-merger maintains the following invariant.

**Invariant** *Each invocation of a $k$-merger outputs the next $k^3$ elements of the sorted sequence obtained by merging the $k$ input sequences.*

A $k$-merger is built recursively out of $\sqrt{k}$-mergers in the following way. The $k$ inputs are partitioned into $\sqrt{k}$ sets of $\sqrt{k}$ elements, which form the input to the $\sqrt{k}$ $\sqrt{k}$-mergers $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_{\sqrt{k}}$ in the left part of the figure. The outputs of these mergers are connected to the inputs of $\sqrt{k}$ ***buffers***. Each buffer is a FIFO queue that can hold $2k^{3/2}$ elements. Finally, the outputs of the buffers are connected to the $\sqrt{k}$ inputs of the $\sqrt{k}$-merger $\mathcal{R}$ in the right part of the figure. The output of this final $\sqrt{k}$-merger becomes the output of the whole $k$-merger. The intermediate buffers are overdimensioned, since each can hold $2k^{3/2}$ elements, which is twice the number $k^{3/2}$ of elements output by a $\sqrt{k}$-merger. This additional buffer space is necessary for the correct behavior of the algorithm, as will be explained below. The base case of the recursion is a $k$-merger with $k = 2$, which produces $k^3 = 8$ elements whenever invoked.

A $k$-merger operates recursively in the following way. In order to output $k^3$ elements, the $k$-merger invokes $\mathcal{R}$ $k^{3/2}$ times. Before each invocation, however, the $k$-merger fills all buffers that are less than half full, i.e., all buffers that contain less than $k^{3/2}$ elements. In order to fill buffer $i$, the algorithm invokes the corresponding left merger $\mathcal{L}_i$ once. Since $\mathcal{L}_i$ outputs $k^{3/2}$ elements, the buffer contains at least $k^{3/2}$ elements after $\mathcal{L}_i$ finishes.

It can be proven by induction that the work complexity of funnelsort is $O(n \lg n)$. We now analyze the cache complexity. The goal of the analysis is to show that funnelsort on $n$ elements requires at most $Q(n)$ cache misses, where

$$Q(n) = O(1 + (n/L)(1 + \log_Z n)) .$$

In order to prove this result, we need three auxiliary lemmas. The first lemma bounds the space required by a $k$-merger.

**Lemma 24** *A $k$-merger can be laid out in $O(k^2)$ contiguous memory locations.*

*Proof.* A $k$-merger requires $O(k^2)$ memory locations for the buffers, plus the space required by the $\sqrt{k}$-mergers. The space $S(k)$ thus satisfies the recurrence

$$S(k) \leq (\sqrt{k}+1)S(\sqrt{k}) + O(k^2) ,$$

whose solution is $S(k) = O(k^2)$. □

In order to achieve the bound on $Q(n)$, the buffers in a $k$-merger must be maintained as circular queues of size $k$. This requirement guarantees that we can manage the queue cache-efficiently, in the sense stated by the next lemma.

**Lemma 25** *Performing $r$ insert and remove operations on a circular queue causes in $O(1+r/L)$ cache misses as long as two cache lines are available for the buffer.*

*Proof.* Associate the two cache lines with the head and tail of the circular queue. If a new cache line is read during a insert (delete) operation, the next $L-1$ insert (delete) operations do not cause a cache miss. □

The next lemma bounds the cache complexity of a $k$-merger.

**Lemma 26** *If $Z = \Omega(L^2)$, then a $k$-merger operates with at most*

$$Q_{\mathrm{M}}(k) = O(1+k+k^3/L+k^3\log_Z k/L)$$

*cache misses.*

*Proof.* There are two cases: either $k < \alpha\sqrt{Z}$ or $k > \alpha\sqrt{Z}$, where $\alpha$ is a sufficiently small constant.

***Case I:*** *$k < \alpha\sqrt{Z}$.* By Lemma 24, the data structure associated with the $k$-merger requires at most $O(k^2) = O(Z)$ contiguous memory locations, and therefore it fits into cache. The $k$-merger has $k$ input queues from which it loads $O(k^3)$ elements. Let $r_i$ be the number of elements extracted from the $i$th input queue. Since $k < \alpha\sqrt{Z}$ and the tall-cache assumption (1.1) implies that $L = O(\sqrt{Z})$, there are at least $Z/L = \Omega(k)$ cache lines available for the input buffers. Lemma 25 applies, whence the total number of cache misses for accessing the input queues is

$$\sum_{i=1}^{k} O(1+r_i/L) = O(k+k^3/L) .$$

Similarly, Lemma 24 implies that the cache complexity of writing the output queue is $O(1+k^3/L)$. Finally, the algorithm incurs $O(1+k^2/L)$ cache misses for touching its internal data structures. The total cache complexity is therefore $Q_M(k) = O(1+k+k^3/L)$.

***Case I:*** $k > \alpha\sqrt{Z}$. We prove by induction on $k$ that whenever $k > \alpha\sqrt{Z}$, we have

$$Q_M(k) \leq ck^3 \log_Z k/L - A(k) , \tag{3.8}$$

where $A(k) = k(1+2c\log_Z k/L) = o(k^3)$. This particular value of $A(k)$ will be justified at the end of the analysis.

The base case of the induction consists of values of $k$ such that $\alpha Z^{1/4} < k < \alpha\sqrt{Z}$. (It is not sufficient only to consider $k = \Theta(\sqrt{Z})$, since $k$ can become as small as $\Theta(Z^{1/4})$ in the recursive calls.) The analysis of the first case applies, yielding $Q_M(k) = O(1+k+k^3/L)$. Because $k^2 > \alpha\sqrt{Z} = \Omega(L)$ and $k = \Omega(1)$, the last term dominates, which implies $Q_M(k) = O(k^3/L)$. Consequently, a big enough value of $c$ can be found that satisfies Inequality (3.8).

For the inductive case, suppose that $k > \alpha\sqrt{Z}$. The $k$-merger invokes the $\sqrt{k}$-mergers recursively. Since $\alpha Z^{1/4} < \sqrt{k} < k$, the inductive hypothesis can be used to bound the number $Q_M(\sqrt{k})$ of cache misses incurred by the submergers. The "right" merger $\mathcal{R}$ is invoked exactly $k^{3/2}$ times. The total number $l$ of invocations of "left" mergers is bounded by $l < k^{3/2} + 2\sqrt{k}$. To see why, consider that every invocation of a left merger puts $k^{3/2}$ elements into some buffer. Since $k^3$ elements are output and the buffer space is $2k^2$, the bound $l < k^{3/2} + 2\sqrt{k}$ follows.

Before invoking $\mathcal{R}$, the algorithm must check every buffer to see whether it is empty. One such check requires at most $\sqrt{k}$ cache misses, since there are $\sqrt{k}$ buffers. This check is repeated exactly $k^{3/2}$ times, leading to at most $k^2$ cache misses for all checks. These considerations lead to the recurrence

$$Q_M(k) \leq \left(2k^{3/2} + 2\sqrt{k}\right) Q_M(\sqrt{k}) + k^2 .$$

Application of the inductive hypothesis and the choice $A(k) = k(1+2c\log_Z k/L)$ yields Inequality (3.8) as follows:

$$\begin{aligned} Q_M(k) &\leq \left(2k^{3/2} + 2\sqrt{k}\right) Q_M(\sqrt{k}) + k^2 \\ &\leq 2\left(k^{3/2} + \sqrt{k}\right) \left[\frac{ck^{3/2}\log_Z k}{2L} - A(\sqrt{k})\right] + k^2 \end{aligned}$$

$$
\begin{aligned}
&\le ck^3\log_Z k/L + k^2\left(1 + c\log_Z k/L\right) \\
&\qquad - \left(2k^{3/2} + 2\sqrt{k}\right) A(\sqrt{k}) \\
&\le ck^3\log_Z k/L - A(k) \ .
\end{aligned}
$$

□

**Theorem 27** *To sort n elements, funnelsort incurs* $O(1+(n/L)(1+\log_Z n))$ *cache misses.*

*Proof.* If $n < \alpha Z$ for a small enough constant $\alpha$, then the algorithm fits into cache. To see why, observe that only one $k$-merger is active at any time. The biggest $k$-merger is the top-level $n^{1/3}$-merger, which requires $O(n^{2/3}) < O(n)$ space. The algorithm thus can operate in $O(1+n/L)$ cache misses.

If $N > \alpha Z$, we have the recurrence

$$Q(n) = n^{1/3} Q(n^{2/3}) + Q_{\mathrm{M}}(n^{1/3}) \ .$$

By Lemma 26, we have $Q_{\mathrm{M}}(n^{1/3}) = O(1+n^{1/3}+n/L+n\log_Z n/L)$.

By the tall-cache assumption (1.1), we have $n/L = \Omega(n^{1/3})$. Moreover, we also have $n^{1/3} = \Omega(1)$ and $\lg n = \Omega(\lg Z)$. Consequently, $Q_{\mathrm{M}}(n^{1/3}) = O(n\log_Z n/L)$ holds, and the recurrence simplifies to

$$Q(n) = n^{1/3} Q(n^{2/3}) + O(n\log_Z n/L) \ .$$

The result follows by induction on $n$. □

This upper bound matches the lower bound stated by the next theorem, proving that funnelsort is cache-optimal.

**Theorem 28** *The cache complexity of any sorting algorithm is* $Q(n) = \Omega(1+(n/L)(1+\log_Z n))$.

*Proof.* Aggarwal and Vitter [6] show that there is an $\Omega((n/L)\log_{Z/L}(n/Z))$ bound on the number of cache misses made by any sorting algorithm on their "out-of-core" memory model, a bound that extends to the ideal-cache model. The theorem can be proved by applying the tall-cache assumption $Z = \Omega(L^2)$ and the trivial lower bounds of $Q(n) = \Omega(1)$ and $Q(n) = \Omega(n/L)$. □

## 3.4 *Distribution sort*

In this section, we look at another cache-oblivious optimal sorting algorithm based on distribution sort. Like the funnelsort algorithm from Section 3.3, the distribution-sorting algorithm uses $O(n \lg n)$ work to sort $n$ elements, and it incurs $O(1+(n/L)(1+\log_Z n))$ cache misses. Unlike previous cache-efficient distribution-sorting algorithms [4, 6, 37, 51, 52], which use sampling or other techniques to find the partitioning elements before the distribution step, our algorithm uses a "bucket splitting" technique to select pivots incrementally during the distribution step.

Given an array $A$ (stored in contiguous locations) of length $n$, the cache-oblivious distribution sort operates as follows:

1. Partition $A$ into $\sqrt{n}$ contiguous subarrays of size $\sqrt{n}$. Recursively sort each subarray.

2. Distribute the sorted subarrays into $q$ buckets $B_1, \ldots, B_q$ of size $n_1, \ldots, n_q$, respectively, such that

   (a) $\max\{x \mid x \in B_i\} \leq \min\{x \mid x \in B_{i+1}\}$ for $i = 1, 2, \ldots, q-1$.

   (b) $n_i \leq 2\sqrt{n}$ for $i = 1, 2, \ldots, q$.

   (See below for details.)

3. Recursively sort each bucket.

4. Copy the sorted buckets to array $A$.

A stack-based memory allocator is used to exploit spatial locality.

The goal of Step 2 is to distribute the sorted subarrays of $A$ into $q$ buckets $B_1, B_2, \ldots, B_q$. The algorithm maintains two invariants. First, at any time each bucket holds at most $2\sqrt{n}$ elements, and any element in bucket $B_i$ is smaller than any element in bucket $B_{i+1}$. Second, every bucket has an associated pivot. Initially, only one empty bucket exists with pivot $\infty$.

The idea is to copy all elements from the subarrays into the buckets while maintaining the invariants. We keep state information for each subarray and bucket. The state of a subarray consists of the index *next* of the next element to be read from the subarray and the bucket number *bnum* where this element should be copied. By convention, $bnum = \infty$ if all elements in a subarray have been copied. The state of a bucket consists of the pivot and the number of elements currently in the bucket.

We would like to copy the element at position *next* of a subarray to bucket *bnum*. If this element is greater than the pivot of bucket *bnum*, we would increment *bnum* until we find a bucket for which the element is smaller than the pivot. Unfortunately, this basic strategy has poor caching behavior, which calls for a more complicated procedure.

The distribution step is accomplished by the recursive procedure DISTRIBUTE$(i, j, m)$ which distributes elements from the $i$th through $(i+m-1)$th subarrays into buckets starting from $B_j$. Given the precondition that each subarray $i, i+1, \ldots, i+m-1$ has its $bnum \geq j$, the execution of DISTRIBUTE$(i, j, m)$ enforces the postcondition that subarrays $i, i+1, \ldots, i+m-1$ have their $bnum \geq j+m$. Step 2 of the distribution sort invokes DISTRIBUTE$(1, 1, \sqrt{n})$. The following is a recursive implementation of DISTRIBUTE:

```
DISTRIBUTE(i, j, m)
1  if m = 1
2     then COPYELEMS(i, j)
3     else DISTRIBUTE(i, j, m/2)
4          DISTRIBUTE(i + m/2, j, m/2)
5          DISTRIBUTE(i, j + m/2, m/2)
6          DISTRIBUTE(i + m/2, j + m/2, m/2)
```

In the base case, the procedure COPYELEMS$(i, j)$ copies all elements from subarray $i$ that belong to bucket $j$. If bucket $j$ has more than $2\sqrt{n}$ elements after the insertion, it can be split into two buckets of size at least $\sqrt{n}$. For the splitting operation, we use the deterministic median-finding algorithm [19, p. 189] followed by a partition.

**Lemma 29** *The median of n elements can be found cache-obliviously using $O(n)$ work and incurring $O(1+n/L)$ cache misses.*

*Proof.* See [19, p. 189] for the linear-time median finding algorithm and the work analysis. The cache complexity is given by the same recurrence as the work complexity with a different base case.

$$Q(m) = \begin{cases} O(1+m/L) & \text{if } m \leq \alpha Z , \\ Q(\lceil m/5 \rceil) + Q(7m/10+6) + O(1+m/L) & \text{otherwise} \end{cases} ;$$

where $\alpha$ is a sufficiently small constant. The result follows. □

In our case, we have buckets of size $2\sqrt{n}+1$. In addition, when a bucket splits, all

subarrays whose *bnum* is greater than the *bnum* of the split bucket must have their *bnum*'s incremented. The analysis of DISTRIBUTE is given by the following lemma.

**Lemma 30** *The distribution step involves $O(n)$ work, incurs $O(1+n/L)$ cache misses, and uses $O(n)$ stack space to distribute n elements.*

*Proof.* In order to simplify the analysis of the work used by DISTRIBUTE, assume that COPYELEMS uses $O(1)$ work for procedural overhead. We will account for the work due to copying elements and splitting of buckets separately. The work of DISTRIBUTE is described by the recurrence

$$T(c) = 4T(c/2) + O(1) .$$

It follows that $T(c) = O(c^2)$, where $c = \sqrt{n}$ initially. The work due to copying elements is also $O(n)$.

The total number of bucket splits is at most $\sqrt{n}$. To see why, observe that there are at most $\sqrt{n}$ buckets at the end of the distribution step, since each bucket contains at least $\sqrt{n}$ elements. Each split operation involves $O(\sqrt{n})$ work. Consequently, the net contribution to the work complexity is $O(n)$. Thus, the total work used by DISTRIBUTE is $W(n) = O(T(\sqrt{n})) + O(n) + O(n) = O(n)$.

For the cache analysis, we distinguish two cases. Let $\alpha$ be a sufficiently small constant such that the stack space used fits into cache.

***Case I,*** $n \le \alpha Z$: The input and the auxiliary space of size $O(n)$ fit into cache using $O(1+n/L)$ cache lines. Consequently, the cache complexity is $O(1+n/L)$.

***Case II,*** $n > \alpha Z$: Let $R(c,m)$ denote the cache misses incurred by an invocation of DISTRIBUTE$(a,b,c)$ that copies $m$ elements from subarrays to buckets. We first prove that $R(c,m) = O(L+c^2/L+m/L)$, ignoring the cost splitting of buckets, which we shall account for separately. We shall argue that $R(c,m)$ satisfies the recurrence

$$R(c,m) \le \begin{cases} O(L+m/L) & \text{if } c \le \alpha L \,, \\ \sum_{i=1}^{4} R(c/2, m_i) & \text{otherwise} \,; \end{cases} \tag{3.9}$$

where $\sum_{i=1}^{4} m_i = m$, whose solution is $R(c,m) = O(L+c^2/L+m/L)$. The recursive case $c > \alpha L$ follows immediately from the algorithm. The base case $c \le \alpha L$ can be justified

as follows. An invocation of DISTRIBUTE$(a, b, c)$ operates with $c$ subarrays and $c$ buckets. Since there are $\Omega(L)$ cache lines, the cache can hold all the auxiliary storage involved and the currently accessed element in each subarray and bucket. In this case there are $O(L + m/L)$ cache misses. The initial access to each subarray and bucket causes $O(c) = O(L)$ cache misses. Copying the $m$ elements to and from contiguous locations causes $O(1+m/L)$ cache misses.

We still need to account for the cache misses caused by the splitting of buckets. Each split causes $O(1+\sqrt{n}/L)$ cache misses due to median finding (Lemma 29) and partitioning of $\sqrt{n}$ contiguous elements. An additional $O(1+\sqrt{n}/L)$ misses are incurred by restoring the cache. As proven in the work analysis, there are at most $\sqrt{n}$ split operations. By adding $R(\sqrt{n}, n)$ to the split complexity, we find the total cache complexity of the distribution step to be $O(L+n/L+\sqrt{n}(1+\sqrt{n}/L)) = O(n/L)$. □

The analysis of distribution sort is given in the next theorem. The work and cache complexity match lower bounds specified in Theorem 28.

**Theorem 31** *Distribution sort uses $O(n \lg n)$ work and incurs $O(1+(n/L)(1+\log_Z n))$ cache misses to sort $n$ elements.*

*Proof.* The work done by the algorithm is given by

$$W(n) = \sqrt{n} W(\sqrt{n}) + \sum_{i=1}^{q} W(n_i) + O(n) ,$$

where each $n_i \leq 2\sqrt{n}$ and $\sum n_i = n$. The solution to this recurrence is $W(n) = O(n \lg n)$.

The space complexity of the algorithm is given by

$$S(n) \leq S(2\sqrt{n}) + O(n) ,$$

where the $O(n)$ term comes from Step 2. The solution to this recurrence is $S(n) = O(n)$.

The cache complexity of distribution sort is described by the recurrence

$$Q(n) \leq \begin{cases} O(1+n/L) & \text{if } n \leq \alpha Z , \\ \sqrt{n} Q(\sqrt{n}) + \sum_{i=1}^{q} Q(n_i) + O(1+n/L) & \text{otherwise} \end{cases}$$

where $\alpha$ is a sufficiently small constant such that the stack space used by a sorting problem of size $\alpha Z$, including the input array, fits completely in cache. The base case $n \leq \alpha Z$ arises when both the input array $A$ and the contiguous stack space of size $S(n) = O(n)$ fit in $O(1+$

$n/L)$ cache lines of the cache. In this case, the algorithm incurs $O(1+n/L)$ cache misses to touch all involved memory locations once. In the case where $n > \alpha Z$, the recursive calls in Steps 1 and 3 cause $Q(\sqrt{n}) + \sum_{i=1}^{q} Q(n_i)$ cache misses and $O(1+n/L)$ is the cache complexity of Steps 2 and 4, as shown by Lemma 30. The theorem follows by solving the recurrence. □

## 3.5 *Conclusion*

This chapter has introduced the notion of cache obliviousness and has presented asymptotically optimal cache-oblivious algorithms for fundamental problems. Figure 3.5 gives an overview of the known efficient cache-oblivious algorithms. Except for matrix addition, Jacobi update [39] and LUP-decomposition [48], all these algorithms are presented in this thesis. For matrix addition, a simple iterative algorithm turns out to be cache-optimal if the matrix elements are read in the same order in which they are stored in memory. The algorithm for LUP-decomposition, due to Toledo [48], uses cache-aware algorithms as subprocedures. By applying the cache-oblivious algorithms presented here, however, his algorithm can be converted into a cache-oblivious one. We shall now look at previous work related to cache obliviousness and cache-efficient divide-and-conquer algorithms. Then, we shall discuss the related work done in proving a gap in asymptotic complexity between cache-aware and cache-oblivious algorithms.

My research group, the Supercomputing Technologies Group at MIT Laboratory for Computer Science, noticed as far back as 1994 that divide-and-conquer matrix multiplication was a cache-optimal algorithm that required no tuning, but we did not adopt the term "cache-oblivious" until 1997. This matrix-multiplication algorithm, as well as a cache-oblivious algorithm for LU-decomposition without pivoting, eventually appeared in [12]. Shortly after leaving the research group, Toledo [48] independently proposed a cache-oblivious algorithm for LU-decomposition with pivoting. For $n \times n$ matrices, Toledo's algorithm uses $\Theta(n^3)$ work and incurs $\Theta(1+n^2/L+n^3/L\sqrt{Z})$ cache misses. More recently, my group has produced an FFT library called FFTW [24], which in its most recent incarnation [23], employs a register-allocation and scheduling algorithm inspired by our cache-oblivious FFT algorithm. The general idea that divide-and-conquer enhances memory locality has been known for a long time [44]. Several researchers [14, 22] have also observed

| Algorithm | Cache complexity | Optimal? |
|---|---|---|
| Matrix Multiplication | $\Theta(m+n+p+(mn+np+mp)/L + mnp/L\sqrt{Z})$ | tight lower bound unknown |
| Strassen's Algorithm | $\Theta(n+n^2/L+n^{\log_2 7}/L\sqrt{Z})$ | tight lower bound unknown |
| Matrix Transpose | $\Theta(1+n^2/L)$ | yes |
| Matrix Addition | $\Theta(1+n^2/L)$ | yes |
| LUP-decomposition [48] | $\Theta(1+n^2/L+n^3/L\sqrt{Z})$ | tight lower bound unknown |
| Discrete Fourier Transform | $\Theta(1+(n/L)(1+\log_Z n))$ | yes |
| Distribution sort | $\Theta(1+(n/L)(1+\log_Z n))$ | yes |
| Funnelsort | $\Theta(1+(n/L)(1+\log_Z n))$ | yes |
| Jacobi multipass filter [39] | $\Theta(1+n/L+n^2/ZL)$ | yes |

**Figure 3-3:** Overview of the known cache-oblivious algorithms.

that recursive algorithms exhibit performance advantages over iterative algorithms for computers with caches.

Recently, Bilardi and Peserico [11] proved that there are problems that are asymptotically harder to solve using cache-oblivious algorithms than with cache-aware algorithms. Their work is based on the HRAM model which is similar to the HMM model [4]. Peserico [38] believes that the result can be extended to the ideal-cache model, too. Although my intuition agrees with him on this account, I feel that the class of cache-oblivious algorithms is interesting and largely unexplored.

CHAPTER 4

# *Cache-oblivious algorithms on other cache models*

In the previous chapters, we have looked at the aspects of algorithm and cache design using the ideal-cache model as a bridging model for caches. The ideal-cache model, however, incorporates four major characteristics that deserve scrutiny: optimal replacement, exactly two levels of memory, automatic replacement, and full associativity. How do cache-oblivious algorithm perform in models without these characteristics? In this chapter we show that optimal cache-oblivious algorithms perform optimally on weaker models, such as multilevel memory hierarchies, probabilistic LRU caches, SUMH [8], and HMM [4]. Consequently, all the algorithmic results in Chapter 3 apply to these models, matching the best bounds previously achieved. Thus, the ideal-cache model subsumes many other cache models, if cache-obliviousness is used as a paradigm for algorithm design.

First, recall the competitiveness result about fully associative LRU caches given in Lemma 2. Let an algorithm $\mathcal{A}$ incur $Q^*(n;Z,L)$ cache misses on a problem of size $n$ using a $(Z,L)$ ideal cache. Then, algorithm $\mathcal{A}$ incurs $Q(n;Z,L) \leq 2Q^*(n;Z/2,L)$ cache misses on a $(Z,L)$ cache that uses LRU replacement.

We define a cache complexity bound $Q(n;Z,L)$ to be ***regular*** if

$$Q(n;Z,L) = O(Q(n;2Z,L)) . \tag{4.1}$$

All algorithms presented in this thesis have regular cache complexities. I believe that most algorithms have regular cache complexity if the input does not fit in the cache. The following lemma shows that LRU is competitive with ideal replacement for the same cache size if the regularity condition is satisfied.

**Lemma 32** *For algorithms with regular cache complexity bounds, the asymptotic number of cache misses is the same for LRU and optimal replacement.*

*Proof.* By the regularity condition (4.1), the number of misses on the $(Z/2, L)$ ideal-cache is at most a constant times more than the number of misses on a $(Z, L)$ ideal-cache. The result follows from Lemma 2. □

The rest of this chapter is as follows. Section 4.1 proves that optimal cache-oblivious algorithms perform optimally on multilevel caches. Section 4.2 contains a similar proof for probabilistic LRU caches. Using 2-universal hashing [36, p. 216], Section 4.3 ports optimal cache-oblivious algorithms to memories with manual replacement, such as SUMH and HMM, without compromising optimality.

## 4.1 *Multilevel caches*

Although models incorporating multiple levels of caches may be necessary to analyze some algorithms, for cache-oblivious algorithms, analysis in the two-level ideal-cache model suffices. Specifically, this section proves that optimal cache-oblivious algorithms also perform optimally in computers with multiple levels of LRU caches.

We assume that the caches satisfy the ***inclusion property*** [28, p. 723], which says that the values stored in cache $i$ are also stored in cache $i+1$ (where cache 1 is the cache closest to the processor). We also assume that if two elements belong to the same cache line at level $i$, then they belong to the same line at level $i+1$. Moreover, we assume that cache $i+1$ has strictly more cache lines than cache $i$. These assumptions ensure that cache $i+1$ includes the contents of cache $i$ plus at least one more cache line.

The multilevel LRU cache operates as follows. A hit on an element in cache $i$ is served by cache $i$ and is not seen by higher-level caches. We consider a line in cache $i+1$ to be ***marked*** if any element stored on the line belongs to cache $i$. When cache $i$ misses on an access, it recursively fetches the needed line from cache $i+1$, replacing the least-recently

accessed unmarked cache line. The replaced cache line is then brought to the front of cache $(i+1)$'s LRU list. Because marked cache lines are never replaced, the multilevel cache maintains the inclusion property. The next lemma, whose proof is omitted, asserts that even though a cache in a multilevel model does not see accesses that hit at lower levels, it nevertheless behaves like the first-level cache of a simple two-level model, which sees all the memory accesses.

**Lemma 33** *A $(Z_i, L_i)$-cache at a given level $i$ of a multilevel LRU model always contains the same cache lines as a simple $(Z_i, L_i)$-cache managed by LRU that serves the same sequence of memory accesses.* □

*Proof.* By induction on the cache level. Cache 1 trivially satisfies the statement of the lemma. Now, assume that cache $i$ satisfies the statement of the lemma.

Assume that the contents of cache $i$ (say $A$) and hypothetical cache (say $B$) are the same up to access $h$. If access $h+1$ is a cache hit, contents of both caches remain unchanged. If access $h+1$ is a cache miss, $B$ replaces the least-recently used cache line. Recall that we make assumptions to ensure that cache $i+1$ can include all contents of cache $i$. According to the inductive assumption, cache $i$ holds the cache lines most recently accessed by the processor. Consequently, $B$ cannot replace a cache line that is marked in $A$. Therefore, $B$ replaces the least-recently used cache line that is not marked in $A$. The unmarked cache lines in $A$ are held in the order in which cache lines from $B$ are thrown out. Again, from the inductive assumption, $B$ rejects cache lines in the LRU order of accesses made by the processor. In addition, $A$ replaces the least-recently used line that is not marked. Thus, cache $i+1$ satisfies the statement of the lemma. □

**Lemma 34** *An optimal cache-oblivious algorithm whose cache complexity satisfies the regularity condition (4.1) incurs an optimal number of cache misses on each level[1] of a multilevel cache with LRU replacement.*

[1]Alpern, Carter, and Feig [8] show that optimality on each level of memory in the UMH model does not necessarily imply global optimality. The UMH model incorporates a single cost measure that combines the costs of work and cache faults at each of the levels of memory. By analyzing the levels independently, our multilevel ideal-cache model remains agnostic about the various schemes by which work and cache faults might be combined.

*Proof.* Let cache $i$ in the multilevel LRU model be a $(Z_i, L_i)$ cache. Lemma 33 says that the cache holds exactly the same elements as a $(Z_i, L_i)$ cache in a two-level LRU model. From Lemma 32, the cache complexity of a cache-oblivious algorithm working on a $(Z_i, L_i)$ LRU cache lower-bounds that of any cache-aware algorithm for a $(Z_i, L_i)$ ideal cache. A $(Z_i, L_i)$ level in a multilevel cache incurs at least as many cache misses as a $(Z_i, L_i)$ ideal cache when the same algorithm is executed. □

## 4.2 Probabilistic LRU caches

This section proves that optimal cache-oblivious algorithms perform optimally on probabilistic LRU caches.

**Lemma 35** *If $\mathcal{A}$ is an optimal cache-oblivious algorithm having regular cache complexity, $\mathcal{A}$ is asymptotically optimal in expectation for probabilistic LRU caches.*

*Proof.* Let $Q_{\text{LRU}}(n_L)$ be the number of cache misses incurred by algorithm $\mathcal{A}$ on an LRU cache with $n_L$ lines. Let $\tau$ be the number of memory operations performed by $\mathcal{A}$. Then, according to Lemma 6, the expected number $\mathrm{E}[Q]$ of cache misses is

$$\mathrm{E}[Q] = \tau P(0) + \sum_{i>0} Q_{\text{LRU}}(i)\,(P(i) - P(i-1)) \; . \tag{4.2}$$

Let $Q_{\text{ideal}}(i)$ be the number of cache misses on an ideal cache with $n_L$ lines. From the definition (4.1) of regularity and competitiveness result Lemma 2 on fully associative LRU caching, we have $Q_{\text{LRU}}(i) \leq k Q_{\text{ideal}}(i)$ for a suitable $k = O(1)$. Therefore, we can upper-bound the expected cache misses of $\mathcal{A}$, given in (4.2) by

$$\mathrm{E}[Q] \leq k\tau P(0) + k \sum_{i>0} Q_{\text{ideal}}(i)\,(P(i) - P(i-1)) \; . \tag{4.3}$$

Let $\mathcal{A}^*$ be the best possible algorithm on the probabilistic LRU cache. We define $Q^*_{\text{LRU}}(n_L)$ as the number of cache misses incurred by algorithm $\mathcal{A}^*$ on an LRU cache with $n_L$ lines, and $\tau^*$ as the number of memory operations performed by $\mathcal{A}^*$. From Lemma 6, the expected number $\mathrm{E}[Q^*]$ of cache misses is

$$\mathrm{E}[Q^*] = \tau^* P(0) + \sum_{i>0} Q^*_{\text{LRU}}(i)\,(P(i) - P(i-1)) \; . \tag{4.4}$$

Let $\mathcal{A}^*$ incur $Q_{\text{ideal}}(n_L)$ cache misses on an ideal cache of size $n_L$. Since ideal caches outperform LRU caches of the same size, $Q_{\text{ideal}}(n_L) \le Q_{\text{LRU}}(n_L)$, and we can lower-bound $\mathrm{E}[Q^*]$, given in (4.4), by

$$\mathrm{E}[Q^*] \ge \tau^* P(0) + \sum_{i>0} Q^*_{\text{ideal}}(i)\,(P(i) - P(i-1))\;. \tag{4.5}$$

Let $W$ and $W*$ denote the work complexities of algorithms $\mathcal{A}$ and $\mathcal{A}^*$, respectively. In the ideal-cache model, each operation accesses the memory $\Theta(1)$ times. Thus, we have $\tau = \Theta(W)$ and $\tau^* = \Theta(W^*)$. Since algorithm $\mathcal{A}$ is an optimal cache-oblivious algorithm, there exists a constant $c$ satisfying $W \le cW*$ and $Q_{\text{ideal}}(i) = cQ^*_{\text{ideal}}(i)$, for all $i$. From (4.3) and (4.5), we have

$$\begin{aligned}
\mathrm{E}[Q] &\le k\tau P(0) + k\sum_{i>0} Q_{\text{ideal}}(i)\,(P(i) - P(i-1)) \\
&\le k\Theta(W)P(0) + k\sum_{i>0} Q_{\text{ideal}}(i)\,(P(i) - P(i-1)) \\
&\le kc\Theta(W^*)P(0) + kc\sum_{i>0} Q^*_{\text{ideal}}(i)\,(P(i) - P(i-1)) \\
&\le kc\Theta(\tau)P(0) + kc\sum_{i>0} Q^*_{\text{ideal}}(i)\,(P(i) - P(i-1)) \\
&= O(\mathrm{E}[Q^*])\;.
\end{aligned}$$

□

## 4.3 *Memories with manual replacement: SUMH and HMM*

In this section, we port optimal cache-oblivious algorithms to caches with manual replacement, without loss of optimality. We shall then apply this technique to SUMH [8] and HMM [4], which are existing memory models with manual replacement.

First, we shall see how a fully associative LRU cache can be maintained in ordinary memory with no asymptotic loss in expected performance. Recall that Chapter 2 presents a method of simulating LRU caches with a conflict-miss overhead. In the following lemma, the simulation technique is competitive to LRU, but it may not be applicable in practice, due to the use of linked list data structures.

**Lemma 36** *A $(Z, L)$ LRU-cache can be maintained using $O(Z)$ memory locations such that every access to a cache line in memory takes $O(1)$ expected time.*

*Proof.* Given the address of the memory location to be accessed, we use a 2-universal hash function [36, p. 216] to maintain a hash table of cache lines present in the memory. The $Z/L$ entries in the hash table point to linked lists in a heap of memory that contains $Z/L$ records corresponding to the cache lines. The 2-universal hash function guarantees that the expected size of a chain is $O(1)$. All records in the heap are organized as a doubly linked list in the LRU order. Thus, the LRU policy can be implemented in $O(1)$ expected time using $O(Z/L)$ records of $O(L)$ words each. □

**Theorem 37** *An optimal cache-oblivious algorithm whose cache-complexity bound satisfies the regularity condition (4.1) can be implemented optimally in expectation in multilevel models with explicit memory management.*

*Proof.* Combine Lemma 34 and Lemma 36. □

**Corollary 38** *The recursive cache-oblivious algorithms for matrix multiplication, matrix transpose, FFT, and sorting are optimal in multilevel models with explicit memory management.*

*Proof.* Their complexity bounds satisfy the regularity condition (4.1). □

In 1990 Alpern et al. [8] presented the uniform memory hierarchy model (UMH), a parameterized model for a memory hierarchy. In the $\text{UMH}_{\alpha,\varrho,b(l)}$ model, for integer constants $\alpha, \varrho > 1$, the size of the $i$th memory level is $Z_i = \alpha\varrho^{2i}$ and the line length is $L_i = \varrho^i$. A transfer of one $\varrho^l$-length line between the caches on level $l$ and $l+1$ takes $\varrho^l/b(l)$ time. The bandwidth function $b(l)$ must be nonincreasing and the processor accesses the cache on level 1 in constant time per access. Figure 4-1 illustrates the UMH model. An algorithm given for the UMH model must include a schedule that, given for a particular set of input variables, tells exactly when each block is moved along which of the buses between caches. Work and cache misses are folded into one cost measure $T(n)$. Alpern et al. prove that an algorithm that performs the optimal number of I/O's at all levels of the hierarchy does not necessarily run in optimal time in the UMH model, since scheduling bottlenecks can occur when all buses are active. In the more restrictive SUMH model [51], however, only one bus is active at a time. Consequently, we can prove that optimal cache-oblivious algorithms run in optimal expected time in the SUMH model.

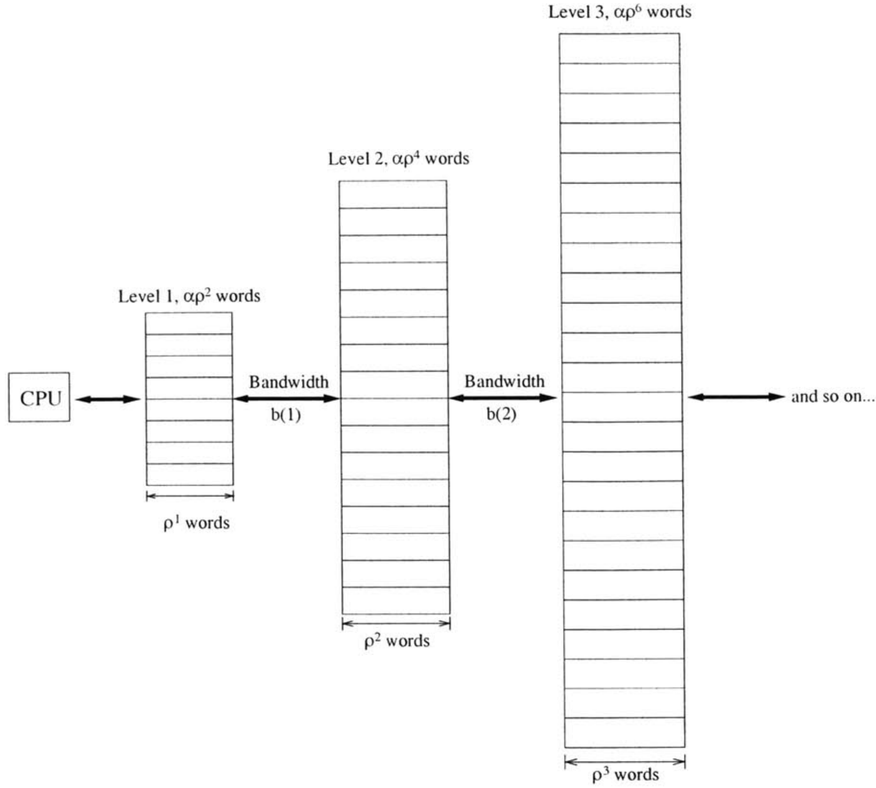



**Figure 4-1:** Illustration of the UMH model showing three levels.

**Lemma 39** *A cache-oblivious algorithm with $W(n)$ work and $Q(n;Z,L)$ cache misses on a $(Z,L)$-ideal cache can be executed in the $SUMH_{\alpha,\varrho,b(l)}$ model in expected time*

$$T(n) = O\Big(W(n) + \sum_{i=1}^{r-1} \frac{\varrho^i}{b(i)} Q(n;\Theta(Z_i),L_i)\Big) ,$$

*where $Z_i = \alpha\varrho^{2i}$, $L_i = \varrho^i$, and $Z_r$ is big enough to hold all elements used during the execution of the algorithm.*

*Proof.* Use the memory at the $i$th level as a cache of size $Z_i = \alpha\varrho^{2i}$ with line length $L_i = \varrho^i$ and manage it with software LRU described in Lemma 36. The $r$th level is the main memory, which is direct mapped and not organized by the software LRU mechanism. An LRU-cache of size $\Theta(Z_i)$ can be simulated by the $i$th level, since it has size $Z_i$. Thus, the number of cache misses at level $i$ is $2Q(n;\Theta(Z_i),L_i)$, and each takes $\varrho^i/b(i)$ time. Since only one memory movement happens at any point in time, and there are $O(W(n))$ accesses to level 1, the lemma follows by summing the individual costs. □

**Lemma 40** *Consider a cache-oblivious algorithm whose work on a problem of size $n$ is lower-bounded by $W^*(n)$ and whose cache complexity is lower-bounded by $Q^*(n;Z,L)$ on an $(Z,L)$ ideal-cache. Then, no matter how data movement is implemented in $SUMH_{\alpha,\varrho,b(l)}$, the time taken on a problem of size $n$ is at least*

$$T(n) = \Omega\Big(W^*(n) + \sum_{i=1}^{r} \frac{\varrho^i}{b(i)} Q^*(n, \Theta(Z_j), L_i)\Big) ,$$

*where $Z_i = \alpha\varrho^{2i}$, $L_i = \varrho^i$ and $Z_r$ is big enough to hold all elements used during the execution of the algorithm.*

*Proof.* The optimal scheduling of the data movements does not need to obey the inclusion property, and thus the number of $i$th-level cache misses is at least as large as for an ideal cache of size $\sum_{j=1}^{i} Z_i = O(Z_i)$. Since $Q^*(n,Z,L)$ lower-bounds the cache misses on a cache of size $Z$, at least $Q^*(n,\Theta(Z_i),L_i)$ data movements occur at level $i$, each of which takes $\varrho^i/b(i)$ time. Since only one movement can occur at a time, the total cost is the maximum of the work and the sum of the costs at all the levels, which is within a factor of 2 of their sum. □

**Theorem 41** *A cache-oblivious algorithm that is optimal in the ideal-cache model and whose cache-complexity is regular can be executed optimal expected time in the $SUMH_{\alpha,\varrho,b(l)}$ model.*

*Proof.* The theorem follows directly from regularity and Lemmata 39 and 40. □

Aggarwal, Alpern, Chandra and Snir [4] proposed the hierarchical memory model (HMM), shown in Figure 4-2, in which an access to location $x$ takes $f(x)$ time. The authors assume that $f$ is a monotonically nondecreasing function, usually of the form $\lceil \log x \rceil$ or $\lceil x^{\alpha} \rceil$.

**Lemma 42** *Consider a cache-oblivious algorithm with $W(n)$ work and $Q(n;Z,L)$ cache misses on a $(Z,L)$ ideal cache. Let $Z_1 < Z_2 < \cdots < Z_r$ be positive integers such that a cache of size $Z_r$ can hold all of the data used during the execution of the algorithm. Then, the algorithm can be executed in the HMM model with cost function $f$ in expected time*

$$O\Big(W(n)f(s_1) + \sum_{i=2}^{r} f(s_i)Q(n;\Theta(Z_i),1)\Big) ,$$

*where $s_1 = O(Z_1)$, $s_2 = s_1 + O(Z_2)$, ..., $s_r = s_{r-1} + O(Z_r)$.*

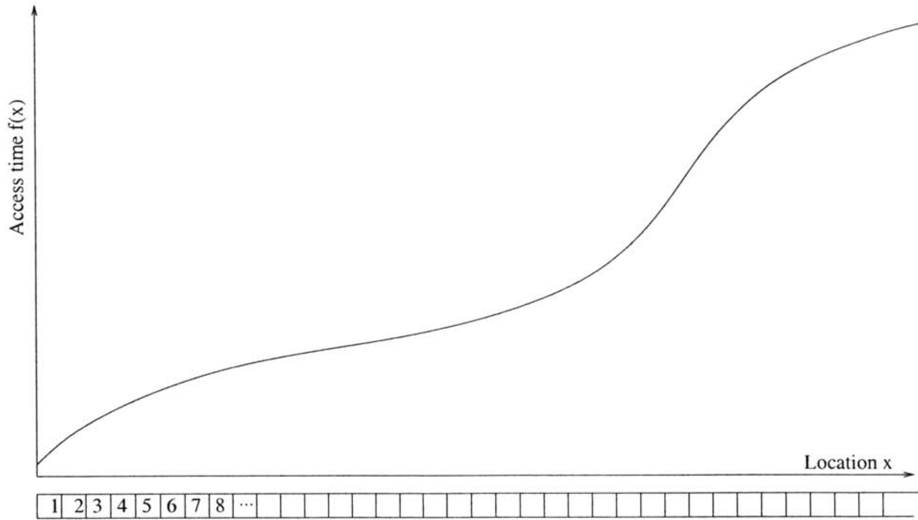



**Figure 4-2:** Illustration of the HMM model.

*Proof.* Using Lemma 36 we can simulate a $\langle(Z_1,1),(Z_2,1),\ldots,(Z_r,1)\rangle$ LRU cache in the HMM model by using locations $1,2,\ldots,s_1$ to implement cache 1, locations $s_1+1,s_1+2,\ldots,s_2$ to implement cache 2, etc. The cost of each access to the $i$th cache is at most $f(s_i)$. Cache 1 is accessed at most $W(n)$ times, cache 2 is accessed at most $Q(n;\Theta(Z_2),1)$ times, and so forth. The lemma follows. □

**Lemma 43** *Consider a cache-optimal algorithm whose work on a problem of size $n$ is lower-bounded by $W^*(n)$ and whose cache complexity is lower-bounded by $Q^*(n;Z,L)$ on an $(Z,L)$ ideal cache. Then, no matter how data movement is implemented in an HMM model with cost function $f$, the time taken on a problem of size $n$ is at least*

$$\Omega\Big(W^*(n)+\sum_{i=1}^{r}(f(Z_{i-1}-1)-f(Z_{i-2}-1))Q^*(n;Z_i,1)\Big)$$

*for any $Z_0=1<Z_1<\cdots<Z_r$ such that a cache of size $Z_r$ can hold all of the data used during the execution of the algorithm.*

*Proof.* The memory of the HMM model can be viewed as a cache hierarchy with arbitrary parameters $Z_0=1<Z_1<\cdots<Z_r$, where the memory elements are mapped to fixed locations in the caches. The processor works on elements in the level 0 cache with $\Theta(1)$ cost. The first $Z_1-1$ elements of the HMM memory are kept in the level 1 cache, the first $Z_2-1$ elements in the level 2 cache, etc. One element in each cache is used as a "dynamic entry" which allows access to elements on higher levels. Accessing a location at level $i$ is

then done by incorporating the memory item in the dynamic element of each of the caches closer to the processor. This "cache hierarchy" obeys the inclusion principle, but it does not do any replacement. As in HMM, memory elements are exchanged by moving them to the processor and writing them back to their new location.

If we charge $f(Z_{i-1}-1)-f(Z_{i-2}-1)$ to a cache miss on cache $i$, an access to element at position $x$ in cache at level $k$ costs $\sum_{i=1}^{k} f(Z_{i-1}-1)-f(Z_{i-2}-1) = f(Z_{k-1}-1)-f(0)$, which is at most $f(x)$. Thus, the access cost for accessing element $x$ is the same in the HMM as in this "cached" HMM model. The cost $T(n)$ of an algorithm in the HMM model can be bounded by the cost of the algorithm in the multilevel model, which is at least

$$\Omega\left(W(n) + \sum_{i=1}^{r} (f(Z_i - 1) - f(Z_{i-1} - 1))Q(n; Z_i, 1)\right) .$$

Since $W(n) \geq W^*(n)$ and $Q(n; Z_i, 1) \geq Q^*(n; Z_i, 1)$, the lemma follows. □

**Theorem 44** *A cache-oblivious algorithm that is optimal in the ideal-cache model and whose cache complexity is regular can be executed in optimal expected time in the HMM model, if the cost function is monotonically nondecreasing and satisfies $f(2x) = \Theta(f(x))$.*

*Proof.* Assume that the cache at level $r$ is big enough to hold all elements used during the execution of the algorithm. We choose $Z_1, \cdots, Z_r$ such that $2f(Z_{i-1} - 1) \leq Z_i - 1 = O(Z_{i-1} - 1)$ for all $1 < i \leq r$. Such a sequence can be computed given that $f$ is monotonically nondecreasing and satisfies $f(2x) = \Theta(f(x))$.

We execute the algorithm as described in Lemma 42 on the HMM model with $2Z_1$, $2Z_2, \ldots, 2Z_r$. The cost of warming up the caches is $\sum_{1 \leq i \leq O(Z_r)} f(i) = \Theta(Z_r f(Z_r))$ which is asymptotically no greater than the cost of the algorithm even if it accesses each input item just once. The result follows from Lemmata 2, 42, and 43. □

## 4.4 *Conclusion*

This chapter showed that cache-oblivious algorithms that are optimal on the ideal-cache model perform optimally on many other platforms provided the regularity condition (4.1) is satisfied. A weakness of our analysis is the assumption of the regularity condition (4.1). Although all algorithms presented in Chapter 3 are regular if the input does not fit in cache,

we can conceive of algorithms that are not regular. Some SPEC95 floating point benchmarks, such as `mgrid` (see Figure 2-9), `hydro2d` (see Figure 2-13), `su2cor` (see Figure 2-14), and `tomcatv` (see Figure 2-15), exhibit sharp decreases in cache misses for certain ranges of the LRU cache size. Halving the cache size considerably increases the cache complexity of such programs. Consequently, the constants involved in the regularity condition (4.1) may turn out to be large, rendering the asymptotic analyses in this chapter loose.

Intuitively, we can see that cache-oblivious algorithms perform well on each level of a multilevel cache. For probabilistic LRU caches, optimal cache-oblivious algorithms perform well for all LRU ranks of the memory location accessed. Similarly in nonuniform memory access cost models such as HMM, cache-oblivious algorithms perform well at all distances of the memory location from the CPU.

CHAPTER 5

# *Conclusion*

This thesis presented an algorithmic theory of caches by proposing the ideal-cache model as a bridging model for caches. In Chapter 2, we discussed the design of practical caches that show near-ideal performance, justifying the ideal-cache model as a "hardware ideal." In Chapter 3, we looked at some efficient cache-oblivious algorithms, whose performance is shown in Figure 3.5, justifying the use of the ideal-cache model as a model for algorithm design. Except for matrix addition, Jacobi update [39] and LUP-decomposition [48], all these algorithms are presented in this thesis. In Chapter 4, we ported optimal cache-oblivious algorithms to more complex memory hierarchies with multiple levels, randomization, and manual replacement, without loss of optimality. Consequently, the ideal-cache model subsumes more complicated models if cache-obliviousness is used as a paradigm for algorithm design. In this section, we shall discuss the myriad of open problems that still need to be addressed to make the ideal-cache model a practically suitable bridging model for caches. We shall now present preliminary results relating to the practical worth of results presented in this thesis.

Hashed line-placement functions are widely recognized as a method of reducing conflict misses. Gonález, Valero, Topham, and Parcerisa [27] claim that "for current workloads and cache organizations, the advantages [of using pseudorandom line-placement functions] can be very large." Currently, the Power PC architecture has an undocumented use of hash functions for line placement [15].

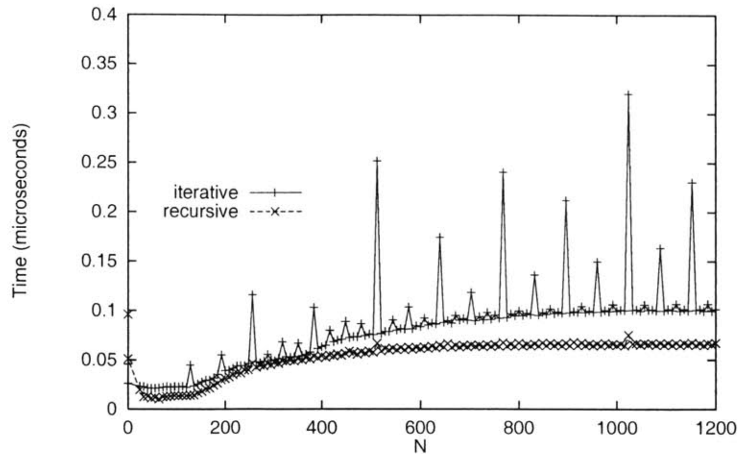



**Figure 5-1:** Average time to transpose an $N \times N$ matrix, divided by $N^2$.

A comprehensive empirical study of cache-oblivious algorithms is yet to be done, however. Do cache-oblivious algorithms perform nearly as well as cache-aware algorithms in practice, where constant factors matter? Does the ideal-cache model capture the substantial caching concerns for an algorithms designer? An anecdotal affirmative answer to these questions is exhibited by the popular FFTW library [23, 24], which uses a recursive strategy to exploit caches in Fourier transform calculations. FFTW's code generator produces straight-line "codelets," which are coarsened base cases for the FFT algorithm. Because these codelets are cache oblivious, a C compiler can perform its register allocation efficiently, and yet the codelets can be generated without knowing the number of registers on the target architecture.

A practical benefit of cache-oblivious algorithms is that they perform well at the register level too. If the base case of a recursive cache-oblivious algorithm is coarsened using inlining, the compiler exploits locality of reference during register allocation. Determining an effective compiler strategy for coarsening base cases automatically is a good research problem.

Figure 5-1 compares per-element time to transpose a matrix using the naive iterative algorithm employing a doubly nested loop with the recursive cache-oblivious REC-TRANSPOSE algorithm from Section 3.2. The two algorithms were evaluated on a 450 megahertz AMD K6III processor with a 32-kilobyte 2-way set-associative L1 cache, a 64-kilobyte 4-way set-associative L2 cache, and a 1-megabyte L3 cache of unknown associativity, all with 32-byte cache lines. The code for REC-TRANSPOSE was the same as presented

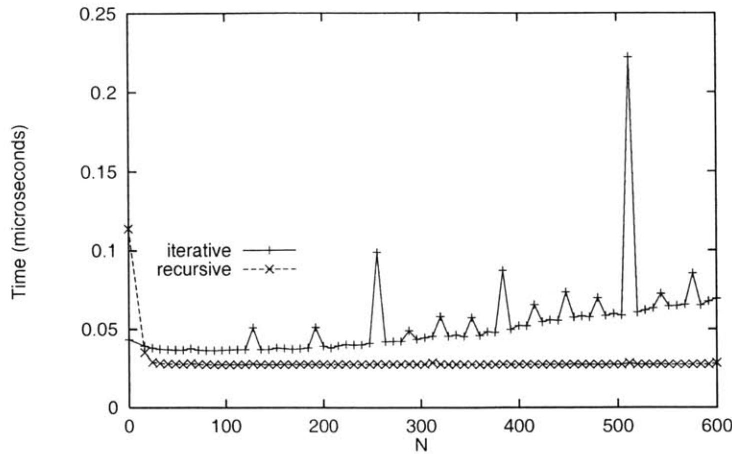



**Figure 5-2:** Average time taken to multiply two $N \times N$ matrices, divided by $N^3$.

in Section 3.2, except that the divide-and-conquer structure was modified to produce exact powers of 2 as submatrix sizes wherever possible. In addition, the base cases were "coarsened" by inlining the recursion near the leaves to increase their size and overcome the overhead of procedure calls.

Although these results must be considered preliminary, Figure 5-1 strongly indicates that the recursive algorithm outperforms the iterative algorithm throughout the range of matrix sizes. Moreover, the iterative algorithm behaves erratically, due to conflict misses caused by bad line placement. Curiously, the recursive algorithm avoids this problem, which I think is due to coarsened base cases. For large matrices, the recursive algorithm executes in less than 70% of the time used by the iterative algorithm, even though the transpose problem exhibits no temporal locality.

Figure 5-2 makes a similar comparison between the naive iterative matrix-multiplication algorithm, which uses three nested loops, with the $O(n^3)$-work recursive REC-MULT algorithm described in Section 3.1. This problem exhibits a high degree of temporal locality, which REC-MULT exploits effectively. As the figure shows, the average time used per integer multiplication in the recursive algorithm is almost constant, which for large matrices, is less than 50% of the time used by the iterative variant. A similar study for Jacobi multipass filters can be found in [39].

I believe that the ideal-cache model opens up many avenues for research in the theory and practice of algorithm and cache design. With sufficient support from the academia and industry, I hope that the ideal-cache model will soon be accepted as a bridging model

for caches.

# Bibliography

[1] AGARWAL, A., HENNESSY, J., AND HOROWITZ, M. Cache performance of operating systems and multiprogramming. *ACM Transactions on Computer Systems 6*, 4 (Nov. 1988), 393–431.

[2] AGARWAL, A., HOROWITZ, M., AND HENNESSY, J. An analytical cache model. *ACM Transactions on Computer Systems 7*, 2 (1989), 184–215.

[3] AGARWAL, A., AND PUDAR, S. D. Column-associative caches: A technique for reducing the miss rate of direct mapped caches. In *20th International Symposium on Computer Architecture* (May 1993), pp. 179–190.

[4] AGGARWAL, A., ALPERN, B., CHANDRA, A. K., AND SNIR, M. A model for hierarchical memory. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing* (May 1987), pp. 305–314.

[5] AGGARWAL, A., CHANDRA, A. K., AND SNIR, M. Hierarchical memory with block transfer. In *28th Annual Symposium on Foundations of Computer Science* (Los Angeles, California, 12–14 Oct. 1987), IEEE, pp. 204–216.

[6] AGGARWAL, A., AND VITTER, J. S. The input/output complexity of sorting and related problems. *Communications of the ACM 31*, 9 (Sept. 1988), 1116–1127.

[7] AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[8] ALPERN, B., CARTER, L., AND FEIG, E. Uniform memory hierarchies. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science* (Oct. 1990), pp. 600–608.

[9] BAILEY, D. H. FFTs in external or hierarchical memory. *Journal of Supercomputing 4*, 1 (May 1990), 23–35.

[10] BELADY, L. A. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal 5*, 2 (1966), 78–101.

[11] BILARDI, G., AND PESERICO, E. Efficient portability across memory hierarchies. Unpublished manuscript, 1999.

[12] BLUMOFE, R. D., FRIGO, M., JOERG, C. F., LEISERSON, C. E., AND RANDALL, K. H. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (Padua, Italy, June 1996), pp. 297–308.

[13] CHATTERJEE, S., JAIN, V. V., LEBECK, A. R., AND MUNDHRA, S. Nonlinear array layouts for hierarchical memory systems. In *Proceedings of the ACM International Conference on Supercomputing* (Rhodes, Greece, June 1999).

[14] CHATTERJEE, S., LEBECK, A. R., PATNALA, P. K., AND THOTTETHODI, M. Recursive array layouts and fast parallel matrix multiplication. In *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (Saint-Malo, France, June 1999).

[15] CHIOU, D. Personal Communication.

[16] CHIOU, D. Extending the reach of microprocessors: Column and curious caching, 1999.

[17] COMPUTER-AIDED AUTOMATION GROUP, LCS, MIT. Malleable Cache Project `http://caa.lcs.mit.edu/caa/mc/index.html`.

[18] COOLEY, J. W., AND TUKEY, J. W. An algorithm for the machine computation of the complex Fourier series. *Mathematics of Computation 19* (Apr. 1965), 297–301.

[19] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. MIT Press and McGraw Hill, 1990.

[20] DUHAMEL, P., AND VETTERLI, M. Fast Fourier transforms: a tutorial review and a state of the art. *Signal Processing 19* (Apr. 1990), 259–299.

[21] DUKE UNIVERSITY. *TPIE User Manual and Reference*. Durham, North Carolina 27706, 1999. Available on the Internet from `http://www.cs.duke.edu/TPIE/`.

[22] FRENS, J. D., AND WISE, D. S. Auto-blocking matrix-multiplication or tracking blas3 performance from source code. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Las Vegas, NV, June 1997), pp. 206–216.

[23] FRIGO, M. A fast Fourier transform compiler. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI)* (Atlanta, Georgia, May 1999).

[24] FRIGO, M., AND JOHNSON, S. G. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing* (Seattle, Washington, May 1998).

[25] FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science* (New York City, New York, 17–19 Oct. 1999), IEEE, pp. 285–297.

[26] GOLUB, G. H., AND VAN LOAN, C. F. *Matrix Computations*. Johns Hopkins University Press, 1989.

[27] GONZÁLEZ, A., VALERO, M., TOPHAM, N., AND M.PARCERISA, J. Eliminating cache conflict misses through xor-based placement functions. In *Proceedings of the 1997 international conference on Supercomputing* (July 1997), pp. 76–83.

[28] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufmann Publishers, INC., 1996.

[29] HILL, M. A case for direct-mapped caches. *IEEE Computer 21*, 12 (Dec. 1988), 25–40.

[30] HILL, M. D., AND SMITH, A. J. Evaluating associativity in cpu caches. *IEEE Computer C-38*, 12 (Dec. 1989), 1612–1630.

[31] HONG, J.-W., AND KUNG, H. T. I/O complexity: the red-blue pebbling game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing* (Milwaukee, 1981), pp. 326–333.

[32] HWANG, K., AND BRIGGS, F. A. *Computer Architecture and Parallel Processing*. McGraw-Hill, Inc., 1985.

[33] JOUPPI, N. P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *17th Annual International Symposium on Computer Architecture* (May 1990), pp. 346–373.

[34] JOUPPI, N. P. Improving direct-mapped cache performance by the addition of a small fully-associative cache refetch buffers. In *25 years of the international symposia on Computer architecture (selected papers)* (June 1998), pp. 388–397.

[35] LAMARCA, A., AND LADNER, R. E. The influence of caches on the performance of sorting. *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (1997), 370–377.

[36] MOTWANI, R., AND RAGHAVAN, P. *Randomized Algorithms*. Cambridge University Press, 1995.

[37] NODINE, M. H., AND VITTER, J. S. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proceedings of the Fifth Symposium on Parallel Algorithms and Architectures* (Velen, Germany, 1993), pp. 120–129.

[38] PESERICO, E. Personal Communication.

[39] PROKOP, H. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, June 1999.

[40] PRZYBYLSKI, S., HOROWITZ, M., AND HENNESSY, J. Performance tradeoffs in cache design. In *Proceedings of the 15th Annual Symposium on Computer Architecture* (June 1988), IEEE Computer Press, pp. 290–298.

[41] PRZYBYLSKI, S. A. *Cache and memory hierarchy design*. Morgan Kaufmann Publishers, Inc., 1990.

[42] RAU, B. R. Pseudo-randomly interleaved memories. In *Proceedings of the International Symposium on Computer Architecture* (1991), pp. 242–246.

[43] SHRIVER, E., AND NODINE, M. An introduction to parallel I/O models and algorithms. In *Input/Output in Parallel and Distributed Computer Systems*, R. Jain, J. Werth,

and J. C. Browne, Eds., vol. 362 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, 1996, ch. 2, pp. 31–68.

[44] SINGLETON, R. C. An algorithm for computing the mixed radix fast Fourier transform. *IEEE Transactions on Audio and Electroacoustics AU-17*, 2 (June 1969), 93–103.

[45] SLEATOR, D. D., AND TARJAN, R. E. Amortized efficiency of list update and paging rules. *Communications of the ACM 28*, 2 (Feb. 1985), 202–208.

[46] SMITH, A. J. Cache memories. *ACM Computing Surveys 14*, 4 (Sept. 1982), 473–530.

[47] STRASSEN, V. Gaussian elimination is not optimal. *Numerische Mathematik 13* (1969), 354–356.

[48] TOLEDO, S. Locality of reference in *LU* decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications 18*, 4 (Oct. 1997), 1065–1081.

[49] VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (Aug. 1990), 103–111.

[50] VITTER, J. S. External memory algorithms. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington* (1998), ACM Press, pp. 119–128.

[51] VITTER, J. S., AND NODINE, M. H. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing 17*, 1–2 (January and February 1993), 107–114.

[52] VITTER, J. S., AND SHRIVER, E. A. M. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica 12*, 2/3 (August and September 1994), 148–169.

[53] WINOGRAD, S. On the algebraic complexity of functions. *Actes du Congrès International des Mathématiciens 3* (1970), 283–288.

[54] YOUNG, N. The *k*-server dual and loose competitiveness for paging. *Algorithmica 11*, 6 (1994), 525–541.