The Effects of Block Size on the Performance of Coherent Caches in Shared-Memory Multiprocessors

Cezary Dubnicki

Technical Report 462
May 1993

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

The Effects of Block Size on the Performance of Coherent Caches in Shared-Memory Multiprocessors

by

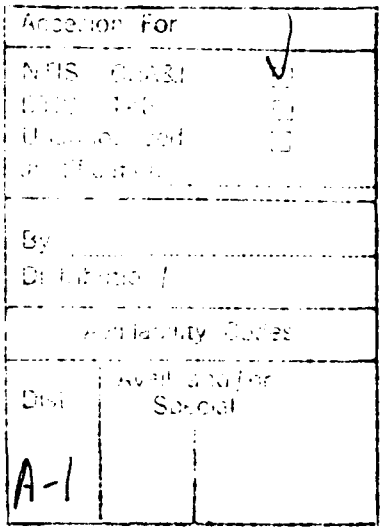
Cezary Dubnicki

Submitted in Partial Fulfillment

of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY



Supervised by

Thomas J. LeBlanc

Department of Computer Science
College of Arts and Science

University of Rochester
Rochester, New York

1993

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1993	technical report / Ph.D. thesis

4. TITLE AND SUBTITLE

The Effects of Block Size on the Performance of Coherent Caches in Shared-Memory Multiprocessors

5. FUNDING NUMBERS

N00014-92-J-1801

6. AUTHOR(S)

Cezary Dubnicki

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Computer Science Dept.
734 Computer Studies Bldg.
University of Rochester
Rochester, New York 14627-0226

8. PERFORMING ORGANIZATION REPORT NUMBER

TR 462.

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Office of Naval Research
Information Systems
Arlington, VA 22217

DARPA
3701 N Fairfax Drive
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

(see page iv)

14. SUBJECT TERMS

parallel programming; shared memory; cache organization; scalable multiprocessors

15. NUMBER OF PAGES

111

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

Curriculum Vitae

Cezary Dubnicki was born in Warsaw, Poland on 2 July, 1961. After finishing high school (Liceum im. K. Gottwalda) in 1980, he enrolled in the Department of Mathematics, Informatics and Mechanics at Warsaw University to study computer science. Cezary graduated with the Magister degree in Computer Science in 1985. His Master's thesis was devoted to an implementation of the distributed programming language *Edison* on a network of microcomputers. This project, supervised by Professor Jan Madey, was joint work with Piotr Kałamjski and Wojciech Wygladała.

After graduation, Cezary accepted a position as a Research Assistant with the Computing Center at Warsaw University. He was employed there from 1985 to 1988, with the exception of 1986, when he severed in the army.

Cezary enrolled in the graduate program in Computer Science at the University of Rochester in the winter of 1989. He pursued research in parallel programming under the direction of Professor Tom LeBlanc and received the Master of Science degree in 1990. In 1993, after completing his Ph.D. degree, Cezary accepted a position as a Research Associate with the Computer Science Department at Princeton University.

Acknowledgments

It is difficult to overestimate the role of my advisor, Tom LeBlanc, in my graduate studies as a whole, and in the preparation of this thesis in particular. His guidance and encouragement were crucial for the progress of my research at Rochester. Most of this dissertation is derived from publications produced jointly with Tom. He helped in defining the adjustable block size cache and provided invaluable input on the contents and organization of this thesis.

Michael Scott, Rob Fowler and Alexander Albicki offered numerous comments on this research, which helped broaden and strengthen its results.

William Bolosky deserves special thanks for providing the traces of parallel applications and their description. I am also indebted to William Bolosky, Mark Crovella, Kai Li, Evangelos Markatos, Jack Veenstra and Jonathan Sandberg for comments on various stages of this work.

I would also like to thank the entire Department of Computer Science, including fellow students and the staff, for the friendly atmosphere which made my graduate studies highly enjoyable.

I wish to thank special friends, Anabelle and Henry Martin, who always made me feel welcome in their home and provided me with invaluable help in organizing my life in Rochester.

Finally, this thesis would not have been possible without my family. The love and support of my wife, Katarzyna, my parents Walerian and Barbara, and my sister Iga, gave me the strength to endure the stress of students life and to overcome all obstacles on the bumpy road to graduation. I feel extremely fortunate to be a member of this family.

This research was supported under NSF CISE Institutional Infrastructure Program Grant No. CDA-8822724, and ONR Contract No. N00014-92-J-1801 (in conjunction with the DARPA HPCC program, ARPA Order No. 8930).

Abstract

Several studies have shown that the performance of coherent caches depends on the relationship between the cache block size and the granularity of sharing and locality exhibited by the program. Large cache blocks exploit processor and spatial locality, but may cause unnecessary cache invalidations due to false sharing. Small cache blocks can reduce the number of cache invalidations, but increase the number of bus or network transactions required to load data into the cache. In this dissertation we use reference traces from a variety of parallel programs and detailed simulation of a scalable shared-memory multiprocessor to examine the effects of cache block size on the performance of coherent caches and quantify this impact with respect to the network bandwidth and latency.

Our results suggest that, regardless of the available bandwidth or latency, applications with good spatial locality favor long cache lines, and for these applications the relative benefits of longer cache lines increase with the bandwidth and latency. For those applications with poor spatial locality, the best choice of cache line size is determined by the product of the network bandwidth and latency, and for these applications, the performance penalty induced by long cache lines increases as this product decreases. We also found that the performance penalty of a mismatch between the cache block size and the sharing patterns exhibited by applications increases with an increase in latency, and decreases with an increase in bandwidth, and can be substantial even on machines with infinite bandwidth.

To reduce this penalty we propose a new cache organization that adjusts the size of data blocks dynamically according to recent reference patterns. In this new cache organization, blocks are split in two when false sharing occurs, and merged back together to exploit spatial locality. Results of simulations in which we varied both the network bandwidth and latency indicate that, for the suite of applications we consider, the adjustable block size cache organization performs better than every fixed block size alternative (including caches that prefetch multiple lines). In addition, the performance benefits of adjustable block size caches increase with an increase in the ratio of network latency to bandwidth. We conclude that adjusting the block size in response to reference behavior can significantly improve the performance of coherent caches, especially when there is variability in the granularity of sharing exhibited by applications.

Table of Contents

Curriculum Vitae ii

Acknowledgments iii

Abstract iv

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Statement of the Problem 1

1.2 Statement of the Thesis 2

1.3 Dissertation Contributions 2

1.4 Organization of the Dissertation 3

2 Related Work 5

2.1 Coherent Caches 5

2.2 Sharing Patterns and Their Effects on Cache Performance 12

2.3 Alternative Solutions 13

2.4 Summary 16

3 Methodology for Cache Evaluation 17

3.1 Architectural Assumptions 17

3.2 Coherency Protocol 18

3.3 Simulation Model 20

3.4 Cache Organizations 21

3.5 Performance Metrics 22

3.6 Memory Reference Traces 25

3.7 Simulation Software 26

4 Evaluation of Fixed Line Size Caches **29**
4.1 Properties of Miss Rate and DTPR 29
4.2 The Effects of Bandwidth 33
4.3 The Effects of Latency 43
4.4 Summary 45

5 Adjustable Block Size Caches **47**
5.1 Overview 47
5.2 Implementation in a Scalable Multiprocessor 49
5.3 Adjusting the Block Size 51
5.4 Integration with the Coherency Protocol 52
5.5 Summary 54

6 Evaluation of Adjustable Block Size Caches **55**
6.1 Cache Instances 55
6.2 Network-Independent Analysis 56
6.3 Effect on *MCPR* 60
6.4 Cache Size 62
6.5 Summary 64

7 Selecting an Instance of the Adjustable Block Size Cache Organization **67**
7.1 Effects of Adjustable Block Size Cache Parameters 67
7.2 High Bandwidth Machines 71
7.3 Medium Bandwidth Machines 74
7.4 Low Bandwidth Machines 75
7.5 High Latency Machines 77
7.6 Summary 79

8 Conclusions and Future Work **81**

Bibliography **85**

A Figures and Tables for All Applications **89**
A.1 *bsort* 90
A.2 *gauss* 92
A.3 *kmerge* 94
A.4 *matmult* 96
A.5 *mp3d* 98

A.6 *pgauss* 100
A.7 *plytrace* 102
A.8 *pmatmult* 104
A.9 *qsort* 106
A.10 *sorbyc* 108
A.11 *sorbyr* 110

List of Tables

3.1 Summary of Reference Traces. 26

4.1 Cache Line Size with Minimal Miss Rate. 30
4.2 Best Fixed Line Size Histogram 42
4.3 Ranges of Preferred Line Sizes 44

6.1 $M(50,10)$ Machine - *MCPR* Relative to $Vblock(4,64,16,(1,1))$ 61

7.1 $M(50,0)$ Machine - *MCPR* Relative to $Vblock(16,64,64,(1,1))$ 72
7.2 $M(50,1)$ Machine - *MCPR* Relative to $Vblock(16,64,64,(1,1))$ 73
7.3 $M(50,5)$ Machine - *MCPR* Relative to $Vblock(8,64,64,(1,1))$ 74
7.4 $M(50,20)$ Machine - *MCPR* Relative to $Vblock(4,64,16,(1,1))$ 76
7.5 Utility of Adjustable Cache as a Function of Latency and Bandwidth. . 77
7.6 *MCPR* of Best Fixed Line Size Cache Relative to $Vblock(16,64,64,(1,1))$. 78

A.1 Split and Merge Statistics - *bsort* 91
A.2 Split and Merge Statistics - *gauss* 93
A.3 Split and Merge Statistics - *kmerge* 95
A.4 Split and Merge Statistics - *matmult* 97
A.5 Split and Merge Statistics - *mp3d* 99
A.6 Split and Merge Statistics - *pgauss* 101
A.7 Split and Merge Statistics - *plytrace* 103
A.8 Split and Merge Statistics - *pmatmult* 105
A.9 Split and Merge Statistics - *qsort* 107
A.10 Split and Merge Statistics - *sorbyc* 109
A.11 Split and Merge Statistics - *sorbyr* 111

List of Figures

3.1 Flow of Messages in the Coherency Protocol 19

4.1 Relationship Between MissR and DTPR - *pgauss* 31
4.2 Relationship Between MissR and DTPR - *plytrace* 31
4.3 Relationship Between MissR and DTPR - *bsort* 32
4.4 Mean Cost Per Reference as a Function of Bandwidth - *bsort* 34
4.5 Mean Cost Per Reference as a Function of Bandwidth - *sorbyr* 36
4.6 Mean Cost Per Reference as a Function of Bandwidth - *pmatmult* . . 37
4.7 Mean Cost Per Reference as a Function of Bandwidth - *matmult* . . . 38
4.8 Mean Cost Per Reference as a Function of Bandwidth - *mp3d* 39
4.9 Mean Cost Per Reference as a Function of Bandwidth - *sorbyc* 40

A.1.1 Miss Rate - *bsort* 90
A.1.2 Data Words Transferred Per Reference - *bsort* 90
A.1.3 Cache Size - *bsort*. 91
A.1.4 Fraction of Block Transfers of Given Size - *bsort* 91
A.2.1 Miss Rate - *gauss* 92
A.2.2 Data Words Transferred Per Reference - *gauss* 92
A.2.3 Cache Size - *gauss*. 93
A.2.4 Fraction of Block Transfers of Given Size - *gauss* 93
A.3.1 Miss Rate - *kmerge* 94
A.3.2 Data Words Transferred Per Reference - *kmerge* 94
A.3.3 Cache Size - *kmerge*. 95
A.3.4 Fraction of Block Transfers of Given Size - *kmerge* 95
A.4.1 Miss Rate - *matmult* 96
A.4.2 Data Words Transferred Per Reference - *matmult* 96
A.4.3 Cache Size - *matmult*. 97
A.4.4 Fraction of Block Transfers of Given Size - *matmult* 97

A.5.1 Miss Rate - *mp3d* 98
A.5.2 Data Words Transferred Per Reference - *mp3d* 98
A.5.3 Cache Size - *mp3d*. 99
A.5.4 Fraction of Block Transfers of Given Size - *mp3d* 99
A.6.1 Miss Rate - *pgauss* 100
A.6.2 Data Words Transferred Per Reference - *pgauss* 100
A.6.3 Cache Size - *pgauss*. 101
A.6.4 Fraction of Block Transfers of Given Size - *pgauss* 101
A.7.1 Miss Rate - *plytrace* 102
A.7.2 Data Words Transferred Per Reference - *plytrace* 102
A.7.3 Cache Size - *plytrace*. 103
A.7.4 Fraction of Block Transfers of Given Size - *plytrace* 103
A.8.1 Miss Rate - *pmatmult* 104
A.8.2 Data Words Transferred Per Reference - *pmatmult* 104
A.8.3 Cache Size - *pmatmult*. 105
A.8.4 Fraction of Block Transfers of Given Size - *pmatmult* 105
A.9.1 Miss Rate - *qsort* 106
A.9.2 Data Words Transferred Per Reference - *qsort* 106
A.9.3 Cache Size - *qsort*. 107
A.9.4 Fraction of Block Transfers of Given Size - *qsort* 107
A.10.1 Miss Rate - *sorbyc* 108
A.10.2 Data Words Transferred Per Reference - *sorbyc* 108
A.10.3 Cache Size - *sorbyc*. 109
A.10.4 Fraction of Block Transfers of Given Size - *sorbyc* 109
A.11.1 Miss Rate - *sorbyr* 110
A.11.2 Data Words Transferred Per Reference - *sorbyr* 110
A.11.3 Cache Size - *sorbyr*. 111
A.11.4 Fraction of Block Transfers of Given Size - *sorbyr* 111

1 Introduction

Caches in uniprocessor systems have been successful in reducing the speed gap between fast processors and much slower memory. For multiprocessors this gap poses an even bigger problem because it is so much wider - due to data sharing, there are usually more levels in the memory hierarchy of a multiprocessor. In particular, a multiprocessor must provide a way to share data between processors, which is usually done by allowing one node to access the local memory of another, or by introducing system-wide shared memory. Both of these methods introduce memory that is non-local to a processor, with delays substantially greater than those for local memory. Therefore it seems natural to use caches in multiprocessor systems to reduce memory latency.

Unfortunately, computers with two or more processors require a technique for ensuring that data remain consistent: all changes to a piece of data must be propagated to all cached copies. In spite of this coherency problem, coherent caches are very popular in modern multiprocessors because they provide the programmer with a hardware implementation of an abstraction of shared memory, which simplifies parallel programming.

1.1 Statement of the Problem

To realize the advantages of a shared-memory model the programmer should be free to forget about peculiarities in the organization of the memory system. The hardware, together with the system software (operating system, libraries, compiler) should allow for the efficient execution of well-written, architecture-independent parallel programs. Unfortunately this is not the case in today's multiprocessor systems.

Several studies have shown that the performance of coherent caches, and in particular the number of invalidations, depends on the relationship between the granularity of sharing and locality exhibited by the program and the cache line size [28; 29; 33]. Long cache lines improve application performance when there is substantial spatial locality (i.e. all words allocated to one cache line are accessed by one processor in some window of time). If the cache line is smaller than the objects used on a per-processor basis, then accessing a single object can generate references to multiple cache lines, increasing misses and decreasing the prefetching effect of spatial locality [33]. However, if the cache line is too large, then *false sharing* [48] is introduced (wherein two data items not being shared happen to reside in the same cache line), which increases the number of

invalidations [29]. Since the granularity of sharing can vary widely among programs, the likelihood of a mismatch between the line size and the grain size used by some programs is high.

One compromise solution is to use short cache lines and prefetching. A short cache line minimizes false sharing, while prefetching reduces the number of transactions required to load an object larger than a cache line into the cache. Unfortunately, prefetching has its limitations as well. The overhead of prefetching cannot be recovered unless the prefetched data is referenced, which may not be the case when the object size used by the program is smaller than the amount of data being prefetched. Also, as with large cache lines, prefetching may increase the number of invalidations, since prefetched data becomes a candidate for invalidation, even if it is never referenced.

1.2 Statement of the Thesis

Although there is no single coherency block size that is optimal for all applications, we claim that we need only a few block sizes to satisfy many different applications, regardless of the available network bandwidth. Moreover, we claim that we can avoid the performance loss caused by a mismatch between the size of the coherency unit and an application's sharing patterns by making the coherency protocol operate on variable-size data blocks instead of fixed-size cache lines. In this scheme, each data block represents a contiguous segment of address space on which coherency is maintained, and the size of each data block is adjusted dynamically, according to recent reference patterns. We propose a new cache organization based on this idea, hereafter referred to as *adjustable block size coherent caches*, which are designed specifically for scalable shared-memory multiprocessors.

1.3 Dissertation Contributions

The first contribution of this dissertation is a thorough analysis of fixed line size cache performance based on extensive simulations of parallel applications. In this study we consider eleven applications with various degrees of spatial and processor locality. We simulate these applications on five multiprocessors with various network bandwidth levels. We identify the relationship between cache performance metrics, like the miss rate, the amount of data transferred per reference, and the mean cost per reference, and the characteristics of an application and the host multiprocessor.

The second contribution is a proposal for adjustable block size caches, which can be used to reduce the penalty caused by a mismatch between the cache line size and an application's sharing patterns. We describe the implementation of this new cache organization in a scalable multiprocessor, and discuss the tradeoffs involved in the implementation. Using simulation, we compare the performance of the new cache organization with existing cache implementations, and quantify the relative advantages and disadvantages of adjustable block size coherent caches. We also identify how the perfor-

mance of each cache implementation depends both on the application and the network latency and bandwidth of the host multiprocessor.

1.4 Organization of the Dissertation

The remainder of the dissertation is organized as follows.

Chapter 2 discusses previous research related to this thesis. It includes a description of coherent caches in bus-based and scalable multiprocessors, and gives an overview of the work on analysis of data sharing patterns and techniques proposed for dealing with the tradeoff between locality and sharing.

Chapter 3 describes the multiprocessor architecture we simulated in our experiments, and the parallel applications used in our studies. It also contains a description of the simulation software, and the performance metrics recorded during each simulation.

Chapter 4 evaluates the performance of fixed line size caches on our application suite. First, we examine the relationship between the miss rate and the amount of data transferred per reference for each program in our application suite. Next we investigate the range of cache line sizes preferred by the applications in our suite, and explain how changes in the network bandwidth and latency influence this range.

Chapter 5 describes the idea of adjustable block size coherent caches, together with the implementation details for a scalable multiprocessor.

Chapter 6 evaluates the performance of adjustable block size caches. Using the results of chapter 4 we propose specific instances of adjustable caches for a number of machines with various network bandwidth levels. Next we compare the performance of selected instances of adjustable block size caches against fixed line size caches with various degrees of multi-line prefetching. In this comparison study we use network-independent metrics, such as the miss rate, the amount of data transferred per reference, and the cache size. Finally we show just how much an adjustable cache can reduce the mean cost per reference for our application suite.

Chapter 7 investigates in detail how the performance of adjustable caches (expressed as the mean cost per reference) depends on specific parameters used in the implementation. We vary the network bandwidth and latency in a series of experiments to show how the performance of adjustable caches depends on the network characteristics of the host multiprocessor. We then show how to select an instance of the adjustable cache organization that is best suited for a particular machine, and quantify how much improvement over the best fixed line size cache we can expect to achieve.

We conclude the dissertation and suggest directions for future work in chapter 8.

2 Related Work

There is a large body of work related to coherent caches in general, and the tradeoff between spatial locality and sharing in particular. We organize our discussion of this work as follows. Section 2.1 describes different ways to implement coherency, including coherent hardware caches, operating system implementations of coherent memory, and other software-based coherence schemes.

In section 2.2 we describe research on the properties of data sharing in parallel programs and their effects on cache performance. We consider the role of the cache line size, and discuss attempts at modeling the behavior of parallel programs.

Section 2.3 contains a discussion of various solutions for dealing with the tradeoff between locality and sharing. We include here cache-sensitive restructuring of program code, software and hardware prefetching, clever implementation of synchronization primitives, and multiple context processors.

Section 2.4 summarizes the related work and distinguishes the work described in this dissertation from previous work.

2.1 Coherent Caches

Hardware solutions to the coherency problem provide data coherency through a coherency protocol built into the hardware, which automatically detects modifications and requests for shared data and ensures that each user sees these data according to the particular data coherency scheme implemented. The details of the implementation of a coherency protocol strongly depend on the architecture.

The two families of shared-memory architectures are bus-based systems and scalable architectures. The former group includes the Sequent Symmetry, Encore Multimax and Silicon Graphics Iris. These machines are usually built around a single bus, to which a limited number of processors (up to 30) may be connected. There is more architectural variation in the latter group, which consists of switching-network based systems like the BBN Butterfly, ring-based systems like the KSR1, mesh-based systems like the MIT Alewife, and multibus machines like the Wisconsin Multicube. Finally there are hybrid architectures, like the Stanford DASH, in which bus-based clusters of processors are connected by an interconnection network.

Most of the bus-based multiprocessors implement coherent caches using broadcast protocols [4; 30]. Given the inherent limitations of bus bandwidth in these machines, there is increasing interest in coherence schemes that do not require a broadcast medium. Recent proposals for scalable coherent caches implement coherency using a directory-based protocol [15; 38]. In this type of protocol, a directory associated with a memory module maintains information about the distribution of copies of data across caches. When write requests are serviced, invalidation messages are sent to the caches containing a copy of the data.

There are also software-based solutions to the coherency problem which are implemented on top of standard uniprocessor caches. Coherency is maintained by software, usually by a smart compiler. Minimal hardware support is assumed, like special read instructions that bypass the cache. Other software-based solutions include operating system implementations of coherent shared memory, where coherency is maintained by allowing only one writer to a page and replicating read-only pages. These implementations use the standard virtual memory mechanism to control page accesses.

Both groups of solutions are important for this dissertation. Software-based solutions introduce new possibilities for avoiding the performance hit caused by a mismatch between the line size and an application's sharing pattern. Hardware implementations are relevant because this thesis proposes a hardware solution to the problem of a mismatch between the line size and an application's sharing pattern.

2.1.1 Coherent Caches in Bus-Based Multiprocessors

Coherent caches in bus-based multiprocessors are usually implemented with a technique called *snooping* [4]. Each processor is connected to the bus and has a snooping module that watches the bus traffic. If a bus transaction affects the data in a processor's cache, the snooping module undertakes the proper action as defined by the protocol.

Snooping is possible because the bus, with its broadcasting capability, provides a system-wide synchronization and communication device. One consequence of a cheap broadcast is that it is always possible to extract the information about copies of a given piece of data by broadcasting a request on the bus. Therefore bus-based systems do not need a directory, which is necessary for scalable multiprocessors. Instead, a processor cache line is usually extended with a few bits that indicate the coherency state of the line. The set of possible states depends on the protocol.

There are three main types of coherency protocols used in bus-based machines. The first category consists of *write-through with invalidate* protocols used in dual-processor machines like the IBM 3033 and the VAX 11/780, and early multiprocessors like the Encore Multimax and Sequent Balance 8000. In this type of protocol each write is propagated to main memory, which results in heavy bus traffic. Since the bus is usually a bottleneck in such systems, this kind of protocol is not very efficient.

The second category consists of *write-invalidate with copy-back* protocols. The key idea here is the concept of ownership of modified data: after the first write to a block of data, the processor performing the write becomes the owner of the block. Subsequent writes from this processor are to the local cache only, until the data is requested by

another processor. Upon such a request, the modified data is usually written back to main memory. This scheme allows for a reduction in bus transactions proportional to the processor locality present in a given application.

Write-invalidate protocols are very popular now, which has lead to a proliferation of variants of this general idea. Ownership by the main memory was introduced in the Synapse N+1 protocol [31]. On a read miss to modified data, there is no cache-to-cache transfer, but instead the data is written back to memory and the requesting cache must reissue the read request. This results in a penalty of three bus cycles for external reads of modified data. In the Berkeley Ownership protocol [35] modified blocks are not written back to main memory on cache-to-cache transfers. Therefore it is possible for data to be in the shared dirty state. The write back takes place when a dirty line is evicted from the owning cache. The Illinois protocol [44] extends the concept of ownership to unmodified data, which avoids unnecessary bus transactions on the first write to local data.

The third category consists of *write-update* protocols. In this scheme each write to shared data is propagated to all copies with one bus transaction. There is a special bus line that is asserted when a write is propagated and there are copies of this data in other caches. If there are no such copies, a writer notices that it has the only copy and subsequent writes go to the local cache until the data is requested by another processor. In this way write-update protocols implement a copy-back policy for blocks that are not shared. This protocol is used in the DEC Firefly [47].

Analysis of sharing patterns of parallel programs indicates that write-invalidate protocols are the best in most cases. This conclusion is supported by the increasing popularity of this type of protocol in the design of new multiprocessors [5].

2.1.2 Coherent Caches in Scalable Multiprocessors

A popular method for maintaining cache coherency in large-scale shared-memory multiprocessors is called the *directory scheme* [13]. In this scheme physical memory is divided into blocks of a fixed size. There is a tag (or directory entry) associated with each block. Tags form a directory. Each entry contains the state of the block (defined by two bits: *modified* and *locked*) and one bit per processor. Bit P is set if this block is cached by processor P. Additionally, each cache entry maintains status bits for the entry (*valid, modified*). A cache entry may be in one of the following states: Valid, Valid-Modified, and Invalid. A memory block may be in one of 4 states:

- *Absent* - no cache holds a copy
- *Read-shared* - one or more caches hold a copy and it is not modified
- *Modified* - exactly one cache holds a copy and it is modified
- *Locked* - an operation on this block is in progress

Two different protocols are used: *write-invalidate* causes invalidation of all copies of data kept in the caches upon a write request; *write-update* updates these copies with a new

value. These protocols are similar to those used in bus-based machines. However, there are significant differences due to the underlying architecture. The complete definition of state transitions and actions taken by a write-invalidate protocol can be found in [13].

The directory scheme is well-suited for large-scale multiprocessors because it does not require broadcast, as do bus-based coherence schemes. It allows for memory and directory distribution to avoid bottlenecks. However, there are two drawbacks to this scheme. First it requires $O(P^2)$ memory, where P is the number of processors. This is because we have to keep a bit for each processor and each memory block, and the total memory size is usually proportional to the number of processors. The second drawback is that the coherence protocol may result in substantial communication overhead, for example when a variable is used in a producer-consumer style (read-write) and a write-invalidate protocol is used.

A lot of work has been done on reducing memory requirements of the original directory scheme, which we will call the *full* directory scheme, to distinguish it from newer versions. One proposal, called *Limited Directories* [3], limits the number of processors that can keep a copy of a datum in their caches. Instead of one bit per processor, now a directory has to keep pointers to a few processors that cache the data. When more than the allowed number of processors wants to cache a particular block, some of the copies are invalidated to make room for new copies. The memory requirement of this scheme is on the order of $P \log P$, because a pointer occupies $\log P$ space. It has been shown [14] that it is possible to achieve good performance with this scheme for many applications. However, there are a few applications for which the results are poor due to directory thrashing caused by data shared by many processors. It is possible to improve the performance even in this case, but at the cost of program transformations.

Another approach limiting the memory requirements of the directory-based scheme is the chained directory [14; 34]. In this scheme a linked list of caches containing the copy of a given data is maintained. The head of the list is usually a directory entry. Double-linked lists may be used to efficiently implement cache replacement. In this scheme write invalidation or update takes time proportional to the length of the list.

A similar idea of keeping the directory information with the caches has been presented in [46]. In this solution the bits indicating where a given copy resides are associated with each cache entry.

Many simulations have shown that parallel programs have relatively few variables that are shared by many processors. However, it is important to support such sharing efficiently. This observation was used in the design of two schemes: *LimitLESS* directories and *Tag* caches. LimitLESS directories [15] were designed for the ALEWIFE multiprocessor system [2]. A limitless directory is a limited directory scheme with the added feature that exceptional sharing of data by many processors is managed in software. ALEWIFE supports this efficiently using very fast context switch. The tag cache scheme implements a directory in the form of two-level caches associated with memory modules. Each memory module has one cache indexed by a memory block, which is able to keep a few (usually one) pointers to processors caching the block. The second cache, much smaller than the first, keeps full directory entries, i.e. one bit per processor. This scheme allows performance quite close to that of full directories, with much lower

memory requirements [43].

One concrete example of a coherent cache implementation for large-scale shared-memory multiprocessors is the DASH project [38]. In this multiprocessor each *cluster* consists of a small number of processors and a portion of the shared memory connected together by a bus. Memory coherency is maintained with a snooping protocol within a cluster and with a distributed write-invalidate protocol across clusters. This distributed protocol uses a presence bit directory, where one bit represents one cluster. The directory is distributed across all clusters - each one has a part of the directory responsible for maintaining coherency of the portion of shared memory associated with a cluster.

There are also other proposals for scalable coherent caches that are not derived explicitly from the directory scheme. An example of such a scheme is cache management in the PLUS system [8]. This scheme is based on page replication. There is a memory coherence manager associated with each node (and its memory). For each virtual page there is a master physical page and replicas. All these pages are arranged in one list, with the master page at the head. On each node there is a replication structure which consists of two tables. The master table contains the global physical page address of each locally replicated page; the next-copy table identifies the successor, if any, along the list. A write-update protocol is used and each write starts with the master copy. Although pages are replicated, the unit of coherence is not a page, but one word. There is a cache of pending writes associated with each memory coherence manager. While a write is in progress, a read to this location from the same processor that issued the write is blocked until the write completes. Note that write-update is essential for this organization; it would not be efficient to replicate pages and maintain coherency of words with a write-invalidate protocol.

The PLUS system is a hybrid scheme because software plays a significant role in cache management. In particular, page replication and migration is done by software. This makes the system similar to the operating system solutions described below. However, hardware support is significant - there are cache coherence managers and special counters that count references from each processor to each page to determine when migration or replication should occur.

2.1.3 Operating System Implementations of Coherent Memory

There is a large body of work on operating system solutions to the memory coherency problem in large-scale multiprocessor systems. This work uses standard virtual memory hardware and implements coherent memory in the operating system layer. It it important to note that in many cases this work is directly relevant to coherent caches. In fact, we can treat operating system solutions as cache coherence protocols with a large cache block size (equal to the page size).

This work has been motivated by NUMA and NORMA architectures. NUMA stands for Non-Uniform Memory Access and denotes multiprocessors with nonuniformity of memory explicitly visible to the programmer (i.e. no coherent caches). However, remote reference is directly supported by the hardware. NORMA stands for No Remote

Memory Access and denotes machines in which only message passing is available for communication between nodes.

Early work in this area reached different conclusions. Solutions based on page migration and periodic examination of reference bits were found to be not much better than good initial placement of data [41]. Other work [10] used replication and migration driven by page faults to minimize memory latency and froze frequently shared pages in global memory to avoid excessive copying. The conclusion was that it is possible to obtain very good results, within 5% of a hypothetical execution with ideal placement of all data in local memories. Encouraging results with a dynamic page placement policy were reported by the PLATINUM project[20]. Unlike [10], page daemons for "defrosting" previously frozen pages were used, which allows for dynamic adjustment of per-page coherence mechanisms according to recent reference behavior.

An extensive study of replication and migration policies is presented in [37]. Seven different applications were run, each with 46 different memory management policies. Slowdown was measured by comparing the execution time of a given application under a given policy to the time it took to execute a NUMA-tuned version of the application. The best "worst-case" policy can be defined as that which causes the smallest slowdown for all seven applications. For these experiments the best worst case policy slowed down one application by 40%, but for others it did very well, even speeding up some of them.

Another study of replication and migration policies compared them against an optimal, off-line policy (computed with future knowledge of references) [11]. This study assumes a fixed size for the hardware coherency unit (a virtual memory page in this case) and a write-invalidate protocol augmented with remote references. This algorithm can also be used to evaluate the performance of coherent caches provided we use a fixed line size organization. Another interesting aspect of this study is its comparison of different architectures. It is concluded that there is no single policy that will be close to the optimal for all possible architectures. For example, freezing pages is a good policy for one machine, in which the cost of a page transfer is high in comparison to remote access. As the cost of page movement decreases, it pays to move a page between processors to use the temporal locality of references to a page.

For NORMA machines, the implementation of coherent memory by the operating system is called distributed shared memory (DSM) and has been proposed by Li and Hudak [40]. As in NUMA machines, this implementation uses the paging mechanisms to control and maintain single-writer and multiple-reader coherence at a page level. There is no support for remote memory access in the hardware however, so it becomes more expensive, as a page fault followed by message passing is needed.

To achieve good performance of parallel applications on machines with operating system implementations of coherent memory, we need to minimize fine-grain and false sharing. This conclusion is reached by all the studies described above, but unfortunately for many programs, false sharing is a serious problem due to large virtual memory pages. This is also one reason why there is no commercial implementation of coherency by the operating system.

2.1.4 Software-Based Cache Coherence

In software-based cache coherence, a compiler analyzes when it is safe to cache shared read-write datum. At the end of this interval, main memory is made consistent and the cached datum is invalidated.

An example of such a scheme is *selective invalidation* [16]. In this scheme accesses to shared variables within a computational unit are classified as always up to date or possibly stale. Three cache instructions are assumed: Memory-Read (used to read possibly stale data), Cache-Read (reads up-to-date copy from a cache), and Cache-Invalidate (invalidates one local cache line by setting a special change bit which is associated with each cache line).

Another software scheme [17] uses version numbers associated with each variable and birth numbers associated with each cache line. A version number is updated at the end of each phase of a program that resulted in a write to this variable. This update is inserted by the compiler and it affects the local version number only. The birth number is set to the current version number on a read miss or the current version number plus one on a write. A reference results in a cache hit only if the current version number of referenced data is not greater than the birth number of the cache line that contains the requested data.

2.1.5 Line Size in Multiprocessor Caches

The cache line sizes for different multiprocessor systems vary wildly.

Both the VMP project [19] and its follow-on effort, ParaDiGM, support long cache lines, up to 512 bytes. The NYU Ultracomputer project [24] uses 32 byte cache lines. In both DASH [38] and Alewife [2] the cache line size is 16 bytes.

The SPUR bus-based multiprocessor uses 32-byte lines. The SGI 4D multiprocessor has two-level caches: the first level has one 4-byte word lines and implements a write-through policy; the second level cache implements a write-invalidate protocol and has 16-byte lines.

The choice of line size in a multiprocessor is affected by a number of factors. Long cache lines can lead to *cache pollution*, which occurs when data fetched into the cache is not used because the cache line is too big. Worse yet, long lines can lead to an increase in cache misses caused by line replacements, because only a limited number of long cache lines can be stored in a given amount of cache space. However, as caches continue to increase in size, this latter consideration is no longer crucial.

Another argument against long cache lines is based on the observation that network contention increases as the square of the message length [1; 36]. Long cache lines do not necessarily result in long messages however. For example, packetizing long lines into small packets may be used to avoid network congestion.

The most important factor influencing the choice of cache line size for a multiprocessor is the expected pattern of sharing in applications. If we expect fine-grain sharing, only small lines should be used. To exploit spatial locality and coarse-grain sharing, longer lines are preferable.

Many studies have shown that no matter which line size is selected, there are always cases that result in severe performance degradation. This happens when expectations about an application's sharing pattern do not agree with the actual behavior of an application. The next section describes such cases, as well as research on the characterization of sharing.

2.2 Sharing Patterns and Their Effects on Cache Performance

An extensive analysis of data sharing in parallel programs was done by Eggers [27; 28; 29]. She developed a *write-run* model of sharing, which is independent of a particular protocol and, to a certain extent, independent of the multiprocessor architecture. A write run is defined as a sequence of write references to a shared address by a single processor, uninterrupted by any other processor. A write run can contain read references as well as writes, but it must start with a write. It ends with the first reference from another processor. Long write runs suggest that data is shared sequentially, whereas short runs indicate fine-grain sharing. The number of external rereads (i.e. reads that do not belong to any write run) may be used as a measure of read sharing.

Based on the notion of write run, Eggers defined a three state finite automaton. The first state represents the same write run, the second state represents a different (new) write run, and the third state represents external rereads. The transitions between states correspond to references that change a write run. By collecting a number of particular transitions we can obtain a full characterization of sharing according to the write-run model. This characterization is architecture independent and was used by Eggers to compare write-invalidate and write-update protocols. Each protocol defines the coherence cost of the transactions of the finite automaton. The comparison was done by computing the total coherency cost for each protocol. This cost is the sum of the costs of each type of transaction, which is in turn a product of the number of transactions recorded for this trace and the cost of one transaction for a particular protocol.

The predictions of the write-run model were compared with the results of detailed simulations. The write-run model was very inaccurate when the actual cache line size was different from the line size used in the analysis. This is because changes in the size of a coherency block produce either a saving or an additional coherency cost, depending on the inter-processor memory access pattern to words within a block, false sharing, and per-processor locality. Eggers concluded that the size of the coherency block has to be included in the model to give realistic results. Therefore, the write-run model is architecture dependent to a certain degree, because we have to know in advance the cache line size before the model data is collected.

The mismatch between cache line size and program sharing patterns has been studied both in the context of bus-based machines and scalable multiprocessors. Eggers and Katz [28; 29] examined the effect of sharing on cache and bus performance using traces from four application programs. They showed that the miss ratios for some parallel

programs increase when the cache line size increases, while for other programs the miss ratio decreases with an increase in the line size. This behavior is caused by the different degree of sharing in the various applications. For those programs exhibiting fine grain sharing, the number of invalidation misses rises with line size; when there is spatial locality on a per-processor basis, invalidation misses decrease with line size. Both false sharing and per-processor locality were shown to be important factors affecting cache performance when a longer line size was used (32 bytes versus 4 bytes). Coherency overhead (expressed in terms of bus cycles needed to maintain coherency) for a write-invalidate protocol increased by over 75% for two applications, and decreased by 48% and 66% for the other two programs.

Simulation results of three parallel applications on the VMP multiprocessor also show that no single line size is best for all programs [19]. The best cache line size for the three programs they considered varied between 32 and 132 bytes.

Gupta and Weber [33] examined the effect of cache line size on the number and size of invalidations in a multiprocessor system with a directory-based cache coherency protocol. With a four-byte line size, most shared writes generated one invalidation. In some cases, larger line sizes (16 and 64 bytes) increased the number of invalidations substantially (over 50% in one case) and the size of invalidations (increasing the percentage of shared writes affecting more than one processor from 13% to 22%). In other cases, an increase in line size reduced the average number of invalidations from 1.77 to 0.34. Once again, these discrepancies were traced to different sharing patterns in the various applications.

2.3 Alternative Solutions

Each of the studies of the impact of the cache line size concluded that a close match between the degree of sharing in an application and the cache line size is desirable. A number of techniques have been proposed to achieve this goal.

One class of techniques is cache-sensitive application restructuring and data reallocation. Another class consists of solutions which assume that the fetch size is different (usually bigger) than the cache line size. This class includes software and hardware prefetching, and two-level caches. A third class includes techniques not directly related to the tradeoff between cache line size and application's sharing pattern. These techniques include contention-free synchronization, hardware support for low-contention synchronization and multiple context processors, all of which may, in some cases, reduce the severity of the problem.

2.3.1 Cache Sensitive Restructuring

One technique to achieve a match between the degree of sharing in an application and the cache line size is to align data objects on cache block boundaries [48]. This technique has several disadvantages, one of which is that it wastes cache space. More important, it wastes bus or network bandwidth to transfer the unneeded portion of a cache line.

This method also requires some knowledge about program semantics to determine the allocation. Therefore we need a smart compiler or user-supplied hints. Finally the size of an object may change dynamically, depending on the number of processors available, scheduling policy, the particular input, or phase behavior of an application.

Two interesting techniques are described in [25]. The first is to group objects according to sharing patterns, i.e. objects that are written by the same processor are allocated close to each other. The second technique, called indirection, replaces a vector of shared objects by a vector of pointers to these objects. Objects themselves are allocated in separate memory blocks, which eliminates false sharing. This approach is especially useful for objects with a dynamic usage pattern (for example, usage depends on the input), because the first technique fails in such cases. After applying both these transformations manually to programs with a significant degree of false sharing, the false sharing miss rate was reduced typically by 50% to 60%, and the total miss rate by 20% to 40% [25]. In spite of these encouraging results there are serious problems with automating these transformations, which was admitted by the authors. First of all, the analysis of aliases must be performed to identify (and possibly change) accesses to reallocated objects. Also, the portion of the code where the parallelism occurs must be identified. Finally, object sharing patterns must be found.

Even better results in improving the performance of a parallel application by restructuring are described in [18]. Two kinds of transformations were applied manually to *mp3d*, a parallel particle simulator for rarefied flow. In the first transformation the global arrays of particle attributes were replaced by an array of instances of the particle class, aligned to cache block boundaries. The second transformation changes the handling of processor task allocation - instead of an allocation around the particles, the allocation is done around the space (i.e. a processor works on particles currently present in its part of the space). The first transformation reduced false sharing drastically, while the second one increased processor locality. The overall improvement in the cache miss ratio was an order of magnitude. This was possible in part because the original program was badly written.

Practical elimination of false sharing is also reported in [7], which uses software caching at the application level. In this approach, shared data is explicitly copied from shared memory to local memory, modified locally, and then copied back to shared memory.

All of these techniques illustrate the importance of programming style. Unfortunately, they do not solve the problem of false sharing, as it is extremely difficult for a compiler to automatically apply these techniques, since many of them exploit knowledge of the semantics of the problem.

2.3.2 Prefetching

Fetching more than one cache line on a cache miss may look like an ideal solution to the problem of a mismatch between the size of a coherency unit and an application's sharing pattern, especially if we keep the cache line short. This is because a small unit

of invalidation (that is, the short cache line) minimizes the effect of false sharing, while the large fetch size allows for the exploitation of processor locality.

To a certain degree this solution was implemented in the SGI bus-based multiprocessor [5], where the fetch size from memory is four (second-level) cache lines, each of which is 16 bytes long. Unfortunately, there are a few important problems introduced by this technique. Since the fetch size is usually constant, we can at best approximate the average case. Programs that exhibit much less or much more processor locality than allowed by the fetch size will still suffer. The large fetch size may also result in wasted network bandwidth if the fetched data is not used. In addition, insertion of many short lines locks up the cache from the processor. To avoid this effect, complicated solutions are needed, like two-level caches. Last but not least, sometimes it is not clear whether we should prefetch. For example, in a scalable multiprocessor if a write request is sent to the home node, and the lines to be prefetched are in the read-only state, and are distributed between caches of many different nodes, it is easy to come up with examples that perform poorly for any decision on prefetching.

A more refined technique than hardware prefetching is software prefetching [12]. Here, a special instruction is generated by the compiler which initiates the prefetch of the required data long before it is needed. Again, if the cache line is short, software prefetching may reduce false sharing, bringing in the necessary data, and hiding the memory latency. The disadvantages of software prefetching include the extra instruction overhead to generate the prefetch. Moreover, we need a sophisticated compiler and good branch prediction to figure out exactly what data will be needed in the future. This is especially difficult in a scalable machine, where delays are large; we would need to start a prefetch very early to hide these delays.

2.3.3 Clever Synchronization

Naive implementation of synchronization operations often results in fine-grain sharing. For example, busy waiting on a lock implemented with *test_and_set* generates a series of writes to the lock location. Much better is a lock implementation based on *test-and-test_and_set*, but still, after a lock is written, invalidations follow, and all waiting processors reread a new lock value in their caches. Only one processor will succeed however, so the activity of the others unnecessarily consumes system resources. Similar problems occur in the naive implementations of barrier synchronization.

However, if synchronization primitives are implemented carefully (see [42]) the fine-grain sharing caused by the use of these primitives is limited. In such cases the affinity of applications for short cache lines is reduced, which may result in the domination of long cache lines. More studies are needed to verify how much fine-grain sharing present in parallel applications is caused by inefficient synchronization primitives.

2.3.4 Multiple Context Processors

Multiple context processors [2] hide latency by switching from one context (thread) to another when a high latency operation (like a cache miss) is encountered. The cost of

a context switch is usually very low in such processors, on the order of a few cycles. This approach may solve the problem of a mismatch between the cache line size and an application's sharing pattern, but only if there are enough contexts to fill the delays. Since a user cares about the completion time, we would need enough contexts in one application. If such a mismatch is present, we need even more contexts, because there are more coherency transactions due to false sharing or poor exploitation of processor locality. Moreover, the increase in network contention caused by such a mismatch would still be a problem.

2.4 Summary

In the studies discussed above, the relationship between program sharing and cache line size was not the primary emphasis. Thus, this relationship has only been examined using a small set of applications and a few line sizes, and program restructuring techniques are the only remedy that has been offered for a poor match between line size and program sharing.

By limiting the focus to the relationship between program sharing and line size, we hope to quantify the effect of a mismatch over a wide range of sharing patterns and line sizes, and quantify the performance benefits of hardware solutions intended to minimize the impact of any disparity. In doing so, we expand the scope of previous studies to examine the relationship between line size, bandwidth, latency, and cache performance.

3 Methodology for Cache Evaluation

This chapter describes the methodology used in our evaluation of coherent caches. Our evaluation is based on trace-driven simulation of the execution of parallel applications on multiprocessors with various cache organizations.

Section 3.1 characterizes the machine architecture used in our studies. The coherency protocol is described in section 3.2. Simulation-specific assumptions about the multiprocessor architecture are discussed in section 3.3. These assumptions extend the multiprocessor organization described in section 3.1. The parameterized notation of various cache organizations explored in this work is given in section 3.4. Metrics used for the comparison between different cache organizations are discussed in section 3.5. The parallel applications used in our study and their memory reference characteristic are described in section 3.6. The simulation software is discussed briefly in section 3.7.

We use the following cache terminology:

- *cache line* - the unit of organization of the processor cache, consists of the data part and the control part (including the address tag and additional control fields like valid bit, modify bit).
- *cache line size* - the size, in 4-byte words, of a cache line.
- *data block* - a contiguous segment of an address space, on which coherency is maintained.
- *directory entry* - an entry corresponding to one data block, used by the coherency protocol.

For fixed line size caches the distinction between data block size and cache line size is not crucial because both sizes are equal. However we need this distinction when we discuss adjustable block size coherent caches in chapter 5.

3.1 Architectural Assumptions

We will consider a multiprocessor architecture consisting of a number of processing nodes connected by an interconnection network. Each processing node contains a processor,

a processor cache, and a memory module. A memory module contains random access memory and a directory used by a coherency protocol. All nodes are equidistant from each other in the network.

This multiprocessor organization has several levels in the memory hierarchy. The first level consists of the local processor caches. Local memory is the second level. The third level includes the caches of all other nodes. The fourth and final level consists of all nonlocal memory modules. There is no global memory equidistant from all nodes in the machine model; each memory module is local to exactly one processor. A range of physical addresses is assigned to the memory module of each node.

Data coherency is maintained in the hardware. A directory protocol [13] is assumed. Although the coherency protocol described in section 3.2 is able to support various types of data consistency [23], sequential consistency is used in the experiments.

3.2 Coherency Protocol

Data coherency in our machine model is implemented with a variant of the write-invalidate, ownership protocol used in the DASH multiprocessor [38]. (Unlike DASH, there are no local clusters of processors connected by a bus.) This protocol was selected because it is efficient in the number of messages required to maintain coherence. Moreover, it is one of the first protocols to be used in a scalable multiprocessor. In what follows we present a brief description of this protocol; more details are given in section 5.4.

Each data block is assigned to a memory module. This assignment is determined by the physical address of a block. All data in a block must reside in one memory module (i.e. data blocks do not span memory modules). The node containing the memory module to which a given data block is assigned is called the *home node* for the data block.

A data block can be in one of three states: *uncached* data is not cached by any processor; *read-shared* (*shared-remote*) data is cached in an unmodified state by one or more processors; *modified* (*dirty-remote*) data is cached in a modified state on a single processor. The processor containing a data block in the *modified* state is called the *owner* of the data block. There are also in-transit states indicating that an operation is in progress on a particular data block.

The flow of protocol messages in four different cases is depicted in Figure 3.1. Each case is determined by the current state of the referenced data block (i.e. *read-shared* or *modified*) and the requested operation (read or write). A circle represents a node and arrows represent messages. The figure depicts the order of messages in time, rather than the paths messages take. Time flows from top to bottom in each diagram. Whenever a node participates in more than one message for a particular transaction, that node is represented by more than one circle in the diagram.

The simplest case is when a node wants to read data and the corresponding data block is in the *read-shared* state. On a cache miss, the requesting node sends a read request message to the home node of the data block. The home node fetches the data

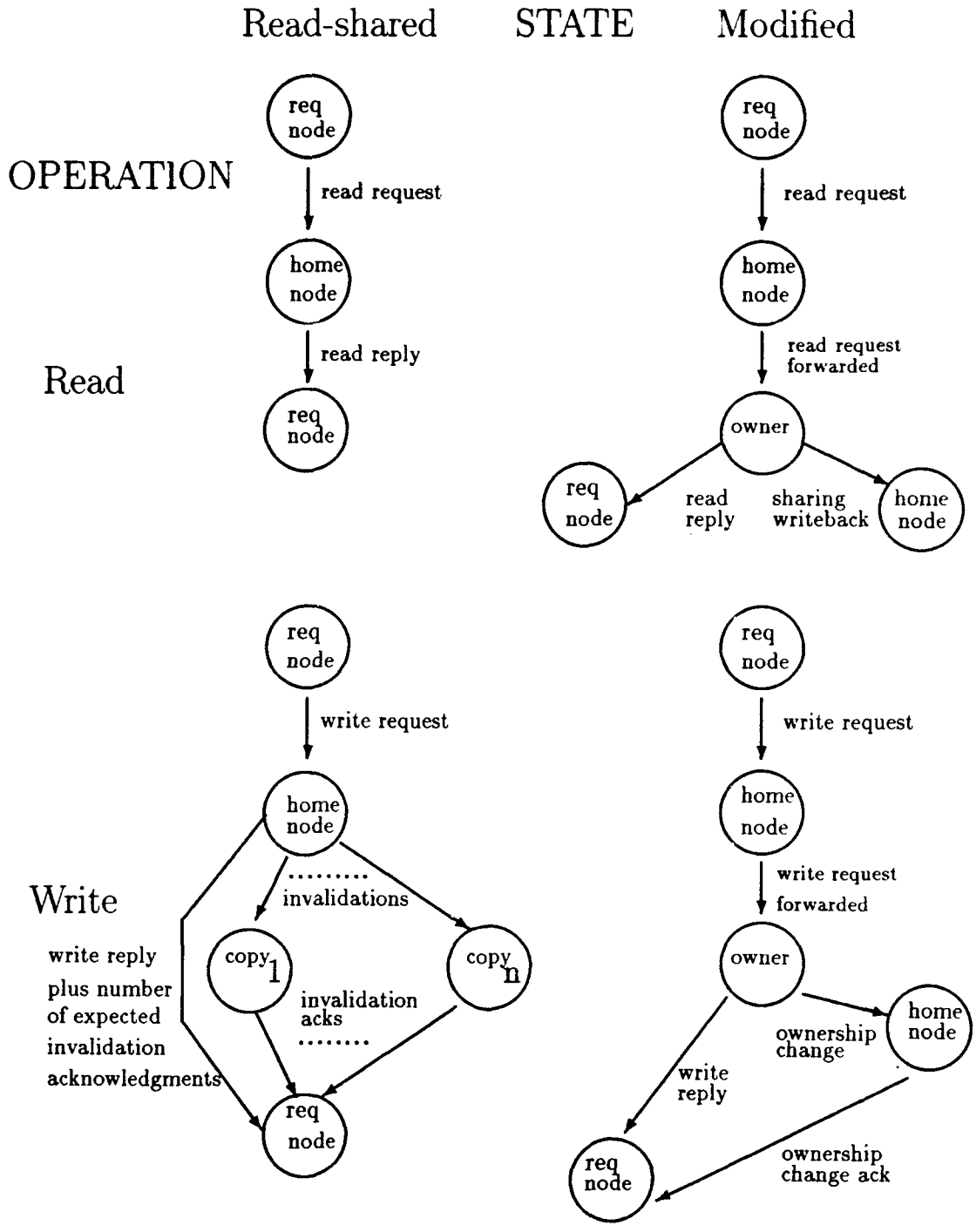


Figure 3.1: Flow of Messages in the Coherency Protocol

from memory, updates its directory information, and returns the requested data in a read reply message.

When the requested data block is in the *modified* state, a read request is forwarded to the owner of this block. The owner provides the requesting processor with the data, and writes the data back to the home node. The new state of the data block is *read-shared.*

On a write request, the home node must invalidate all copies (if any) of the requested data. In the *read-shared* state, the home node immediately sends the data to the requesting node. This message also contains the number of invalidation acknowledgements the requesting node should receive before the write can be regarded as having been performed globally. Next, the home node sends invalidations to all nodes containing copies of the data block. Those nodes invalidate their copies and send invalidation acknowledgements to the original requesting node. The new state of the data block is *modified.*

This protocol admits the use of weak consistency [22; 23], wherein a node can proceed immediately after it receives the data. On a synchronization operation, the processor is stalled until all outstanding writes are performed globally (i.e. until all expected invalidation acknowledgements arrive). Even when sequential consistency is required, this protocol is preferred because it is efficient in the number of messages sent; invalidated nodes send acknowledgements directly to the requesting node, which limits the involvement of the home node.

When a write to modified data is requested, the home node forwards the write request to the owner node. This node sends the data to the requesting node indicating that one acknowledgement should arrive. The owner then notifies the home node about the change in ownership. The home node in turn acknowledges completion of the write transaction. This extra acknowledgement is needed to avoid problems that could arise when the requesting node evicts the cache line before the home node receives the ownership change.

3.3 Simulation Model

It is difficult to fully evaluate a new cache organization with a fixed computer architecture. First, there are many different multiprocessors available on the market now or in the research stage. For scalable multiprocessors there is no single architectural model followed by all designers. Instead we can observe a wide variety of different architectures distinguished by such factors as processor and memory speed, number of processors, and network configuration.

A second factor limiting the use of a fixed architecture model is the rapid progress in hardware. Not long ago 100 MHz processors and 400Mb/sec memory bandwidth were well beyond reach. Today DEC Alpha chips and RAMBUS technology let us achieve or surpass these specifications.

Because of these two factors we decided to parametrize the multiprocessor machine described in section 3.1. This parametrization assumes a scalable multiprocessor with coherent caches, but it is independent of the particular cache organization. No particular

interconnection network is assumed, except that all nodes are equidistant and there is no contention in the network. The parametrization is concerned primarily with the normalized time cost (used in computing the mean cost per reference, see section 3.5) of message transmission between nodes. We assume that all machines use the same processor. The time to satisfy a memory reference from the cache is normalized to 1. The time required for a message transmission of length X (in 4 byte words) between any two nodes is given by a linear function (F stands for *Factor*)

$$F_{latency} + F_{bandwidth} * X$$

where $F_{latency}$ is the network latency contribution, or the normalized time to send a zero-length message, and $F_{bandwidth}$ is the network bandwidth contribution, or the incremental cost of sending one more word of data. Note that as the bandwidth factor $F_{bandwidth}$ increases, the bandwidth of the network decreases.

We assume that the normalized memory latency time is 5, i.e. it takes 5 time units to fetch the first word from memory. We assume that memory bandwidth is always equal to the network bandwidth. In effect fetching from memory contributes only the memory latency to the total cost (provided there is network transmission), regardless of the actual length of a fetched line.

Our simulations assume a machine with 8 processors (due to available traces), local reference time normalized to 1, and an interconnection network characterized by $F_{latency}$ and $F_{bandwidth}$. Again, the coherent cache implementation is not specified, which allows for experiments with different cache organizations.

A particular machine is denoted by

$$M(F_{latency}, F_{bandwidth})$$

Sometimes we use a star ($*$) instead of $F_{bandwidth}$ to denote a group of machines with particular $F_{latency}$ and various levels of bandwidth.

As shown in section 3.5, for any two caches the order established by their performance for a given application P is the same for all machines with the same ratio of $F_{latency}$ to $F_{bandwidth}$. Therefore we use that ratio as a characterization of a class of machines if we want to order various caches according to their performance on a given application.

3.4 Cache Organizations

The adjustable block size coherent caches (introduced in detail in chapter 5) are compared with more traditional, fixed line size coherent caches with various degrees of prefetching. To eliminate the effect of cache size and to simplify the simulations, we assume that all caches are big enough to avoid capacity misses completely. As was shown in [26], the effect of cache size on coherency overhead is minimal, especially when compared with the effect of cache line size. All caches are fully associative, so we do not deal with conflict misses.

The simplest organization, a fixed line size cache without any prefetching is denoted by

$$Fixed(LineSize)$$

where line size is given in 4 byte words.

We also consider an aggressive implementation of prefetching. A cache with a fixed line size of *LineSize* words and prefetching fetches several *LineSize* word data blocks at once, but maintains coherence on each N word block individually. This organization is referred to as

$$Prefetch(LineSize, NumBlocks)$$

where *NumBlocks* is the maximum number of data blocks that can be prefetched on a cache miss (including the requested one). Blocks can be prefetched in both *read-shared* and *write-modified* states. However, for a given prefetch operation, all blocks have to be in the same state, equal to the state of the referenced block. That is, blocks are prefetched up to the first block whose state is different from the state of the referenced block. Moreover, prefetching is possible only if there is an agreement between the state of the referenced block and the reference operation (i.e. *read-shared* and *read* , or *modified* and *write*). Additionally, for prefetching in the *modified* state, all blocks have to be owned by the same node (to avoid excess network messages). Prefetching begins with the first block after the requested one.

Over a wide spectrum of parallel applications, prefetching should offer better performance than a simple fixed line size cache because it is more flexible. That is, a prefetching cache has more options than a non-prefetching cache with respect to how much data it fetches. Additionally, a prefetching cache uses more knowledge related to the sharing behavior of an application because the decision of how much data should be fetched is based on inspection of the coherency states of multiple blocks adjacent to the requested one. If an application exhibits fine-grain sharing, the number of prefetched blocks should be close to zero, and a prefetching cache emulates a fixed line size cache with line size equal to *LineSize*. On the other hand, if there is sufficient spatial locality to be exploited in a parallel application, prefetching should take place, and the cache organization emulates a fixed line size cache with line size equal to $NumBlocks * LineSize$.

The notation for the adjustable block size cache introduced in this thesis is given in chapter 5.

3.5 Performance Metrics

Traditionally the miss rate (*MissR*) has been the metric of choice for quantifying the performance of uniprocessor, as well as multiprocessor, caches. Unfortunately the miss rate is not sufficient as a performance metric in our case for a number of reasons. First of all, in a scalable multiprocessor the time needed to satisfy a miss is not constant, because requested data may reside in many different levels of the memory hierarchy. For example, it may be available in the memory of the requesting node or in the memory of a remote node. The cost will differ in these two cases by at least twice the network latency. Another reason the miss rate does not capture the performance effects of interest is that

the cache block size is not constant in adjustable caches, so the cost of a miss depends on the size of a block containing the requested block. In spite of these problems we consider the miss rate a metric worth considering because it gives an upper bound on how much network latency matters for a particular cache organization.

The cache size (*CacheS*) metric gives us the minimum size of the cache needed for conflict-free (no conflict and capacity misses) execution of a given parallel application. We assume here that cache lines invalidated on coherency transactions can be reused. For the purposes of this metric, the cache size includes not only the data part of the cache, but also the tag part and the additional control fields of a particular cache organization (for example, modify and valid bits). As a result this metric depends on the cache organization, even if we assume that all caches are fully associative.

We define the data transferred per reference (*DTPR*) metric to be the total number of data bytes transferred during the execution of a parallel program (excluding message headers) divided by the number of references. Just as the miss rate captures the effect of network latency, this metric captures the effect of network bandwidth; the lower the bandwidth, the greater the impact of *DTPR* on total execution time.

We define the mean cost per reference (*MCPR*) metric to be the normalized time (in units of local cache accesses) a processor has to wait for data in the absence of contention. (A similar metric was used in [11]). For data requests satisfied by the local cache, the *MCPR* is equal to 1. In the case of a miss, we assume that a processor is stalled till all fetched data arrives. For data requests, the *MCPR* includes the time needed for memory fetches, network transmission, and directory operations. There are also additional costs of block adjustment taken into account in computing the *MCPR* metric for adjustable caches (see chapter 5). These costs are described in detail in section 5.3. However, we do not charge anything for handling multiple cache lines in the case of prefetching caches (except for the cost of increased bandwidth consumption due to fetching more data). As a result the mean cost per reference for prefetching caches is somewhat optimistic.

To compare the performance of two caches we will use their *relative difference* defined as the absolute difference between their *MCPR* values divided by the lower *MCPR* of the two.

It is important to note that the miss rate and *DTPR* metrics do not depend on network latency or bandwidth. *MCPR* **does** depend on these factors, and can be viewed as a combination of the miss rate and *DTPR*, weighted by network latency and bandwidth. That is, for each miss, we incur a delay of at least twice the network latency, and for each word transferred we consume network bandwidth. [1]

MCPR can be expressed as follows:

$$MCPR = 1 + COPR \tag{3.1}$$

where $COPR$ is the coherency overhead per reference. We can compute the overhead for each type of coherency transaction (e.g., a read or write access to read-shared data),

[1]Since we assume sufficiently large caches to avoid capacity misses, there are no misses to local memory other than those associated with a cold start.

sum this overhead over all references, and divide by the total number of references, to arrive at COPR. For example, the coherency overhead associated with a read miss of read-shared data (ignoring the overhead due to initiating a fetch from memory) is:

$$CO(read, read\_shared) = 2 * F_{latency} + CacheLineSize * F_{bandwidth} \quad (3.2)$$

which includes the network round-trip time, and the overhead of sending a full cache line through the network. Similar equations can be used to define the overhead for each coherency transaction. For each such equation, we observe that if we multiply $F_{latency}$ and $F_{bandwidth}$ by the same constant k, then the overhead for each coherency transaction increases by a factor of k. This fact allows us to formulate the following scaling property:

Property 1 *Scaling Property: Let* $COPR(P, C, M(F_{latency}, F_{bandwidth}))$ *denote the coherency overhead per reference for execution of program P on machine* $M(F_{latency}, F_{bandwidth})$ *with cache C. Then, for any cache C, any program P, and any positive constant k, the following holds:*

$$COPR(P, C, M(k * F_{latency}, k * F_{bandwidth})) = k * COPR(P, C, M(F_{latency}, F_{bandwidth}))$$

That is, if we scale both $F_{latency}$ and $F_{bandwidth}$ by the same constant k, then the total coherency overhead per reference scales by the same factor.

Given the *COPR* for one machine, and the scaling factor for another machine with the same ratio of $F_{latency}$ to $F_{bandwidth}$, we can compute *COPR* and *MCPR* for the new machine using the scaling property. From the scaling property we can also infer that the relative performance of two cache organizations on the same application expressed as the ratio of their coherency overheads does not change if we scale the network characterization.

When comparing the performance of two caches, we are most interested in the ratio of their *MCPRs*, not *COPRs*. We note here that the relative performance of two caches expressed as the ratio of *MCPR* does change if we scale the network characterization, which follows from equation 3.1 and the scaling property. However, if cache C_1 offers better performance (i.e. lower *MCPR*) than cache C_2 for program P executed on a given machine M, then cache C_1 is always better than C_2 for program P executed on any machine with the network characterization scaled from M by any factor $k > 0$. Again we can prove this fact using the scaling property and equation 3.1. If we scale both $F_{latency}$ and $F_{bandwidth}$, then what changes is the relative difference in cache performance. If $k > 1$, then this difference increases; for $0 < k < 1$, it decreases.

Recall that $F_{bandwidth}$ is inversely proportional to the network bandwidth. Therefore the *product* of network latency and bandwidth determines the order of caches according to their performance for a given application P. As a consequence, a cache that delivers the best performance among a group of caches for a given application and a given machine remains the best cache for that application and all machines with the same product of network latency and bandwidth. A similar result on the independence of optimal memory placement with respect to scaling of machine characterization is reported in [9].

3.6 Memory Reference Traces

We use memory reference traces to drive our simulations. These traces were collected on an 8-node IBM ACE and in addition to our work, have been used by others to study memory management policies and the performance benefits of disabling coherence to minimize short-term false sharing [11].

The programs in our trace suite fall into three different classes:

- *C-threads applications* - The majority of the programs were implemented using the C-threads package on a non-uniform memory access (NUMA) multiprocessor. *Sorbyr* and *sorbyc* implement red-black successive over-relaxation. The two programs differ in the order of their inner loops: *sorbyr* operates on rows, while *sorbyc* operates on columns. *Plytrace* is a scene rendering program. *Matmult* implements matrix multiplication. Both *bsort* and *kmerge* perform a merge sort; in *bsort* half the processors drop out in each phase, whereas in *kmerge* all processors are busy until the end. *Gauss* is a Gaussian elimination program. All of these programs exhibit coarse-grain parallelism and good processor locality, however spatial locality differs widely among these programs.

- *SPLASH application* - From the SPLASH suite [45] we took one application, *mp3d* which solves a problem in fluid flow simulation. This program has been selected because it exhibits extremely large amount of sharing and very poor spatial locality.

- *Presto applications* - The following programs were implemented in Presto [6]: *qsort* implements parallel quicksort, *pgauss* is a Gaussian elimination program, *pmatmult* is a matrix multiplication program. Like *mp3d*, these applications were written for a bus-based multiprocessor. The first two programs exhibit fine-grain sharing and poor processor locality. *Pmatmult* performs much better than other *Presto* programs.

Each of these programs was executed on the IBM ACE and detailed memory reference traces were collected. The number of memory references made by each program and the fraction of writes is presented in Table 3.1. For each application we give the number of references which are not explicit synchronization references (i.e. lock tests and test-and-sets were deleted from the trace).

As we can see, (table 3.1) both the total number of references and the faction of writes varies widely among the programs.

Unlike in our earlier work on adjustable caches [21] we decided to eliminate synchronization references from the traces. This is because we believe that synchronization seriously distorts cache behavior if not treated specially. For example, synchronization introduces a lot of fine-grain sharing if remote spinning is allowed and spin-locks are grouped together, which in turn overly favors short cache lines. In scalable multiprocessors, synchronization should bypass cache coherence [42] or be explicitly integrated into it as a special case [32]. In either case the effect of synchronization references on our metrics would be minimal.

Table 3.1: Summary of Reference Traces.

Application	Millions of References	Fraction of Writes
bsort	23.62	0.36
gauss	269.70	0.17
kmerge	10.92	0.35
matmult	4.641	0.03
mp3d	19.58	0.42
pgauss	21.67	0.34
plytrace	15.32	0.34
pmatmult	6.81	0.04
qsort	17.53	0.32
sorbyc	104.40	0.21
sorbyr	103.90	0.21

3.7 Simulation Software

We use a two-stage simulator to compare cache organizations for a wide range of host machines. Our simulator consists of two programs (stages). The first stage takes as input an address trace of a parallel application and a particular cache organization (adjustable or fixed, with or without prefetching) with specified parameters. The output of this stage is an intermediate representation of the execution of the program represented by the input trace on a (partially specified) machine with the input cache. Both bandwidth and latency of the host machine are unspecified. The intermediate representation records the number of coherency transactions of a given type encountered during the simulation, where types are defined according to a particular cache organization. For example, one of the transition types for adjustable caches is the transition from the *modified* state to the *read-only* state, where we record the type of adjustment (if any) and the size of the affected blocks. For prefetching caches, a transaction type includes the number of blocks prefetched. Broadly speaking, a transaction type is defined by a relation, so the intermediate representation produced by the first stage can be seen as a relational database.

Apart from this intermediate representation the first stage produces architecture-independent statistics like the miss and data transfer rates, and the amount of cache space required.

The second stage of the two-stage simulator takes as input the intermediate representation produced by the first stage and the specification of a machine (in the form of $F_{latency}$ and $F_{bandwidth}$ factors), together with other costs, like the cost of block adjustment, and the memory bandwidth. The output of the second stage is the mean cost per reference metric.

The first stage takes quite a long time to execute, on the order of several minutes up to an hour, depending on the trace and the machine executing simulations. The second stage, on the other hand, is completed within a few seconds. This simulator design

allows for time and space efficient research of the multi-dimensional parameter space of cache organizations and machines.

4 Evaluation of Fixed Line Size Caches

This chapter evaluates the performance of fixed line size caches. First, we examine the relationship between the miss rate and the amount of data transferred per reference for fixed line size caches and our application suite. Next we investigate the effects of bandwidth and latency on the performance of fixed line size coherent caches.

4.1 Properties of Miss Rate and DTPR

In this section we examine the values of *MissR* and *DTPR* for the programs in our application suite, and identify any dependencies between these two metrics. We also describe how a change in the cache line size affects these metrics.

In simulations in which we varied the cache line size between 1 and 128 words, the minimum miss rate varied widely among applications. The minimum miss rate was highest for *mp3d* (6.7%), *pgauss* (2.7%), and *qsort* (2.1%). All of the other applications exhibited a minimum miss rate below 1% (i.e., *plytrace* 0.64%, *matmult*, 0.34%, *sorbyc* 0.27 *pmatmult* 0.23%, *kmerge* 0.1%, *bsort* 0.08%, *gauss* 0.06%, and *sorbyr* 0.03%). The minimum data transferred per reference was also highest for *mp3d* at 0.214 words per reference. *DTPR* for the other programs was below 0.1 words per reference (i.e., *pgauss* 0.072, *qsort* 0.065, *plytrace* 0.035, *matmult* 0.034, *sorbyc* 0.006, *pmatmult* 0.033, *kmerge* 0.067, *bsort* 0.042, *gauss* 009 and *sorbyr* 0.006). There is a positive correlation between a low minimum miss rate and a low minimum *DTPR* for nearly all programs; however, three programs exhibit a negative correlation: *bsort* and *kmerge* have both low minimum miss rate and a relatively high minimum *DTPR*, while *sorbyc* has a relatively high minimum miss rate and a very low minimum *DTPR*.

For a given application, there is always some line size that minimizes the miss rate. An increase in line size may cause an increase in miss rate, a sure sign of false sharing (if we neglect cache size related effects like cache pollution). Although false sharing affects all applications, some applications exhibit false sharing only with very long cache lines. Table 4.1 gives the line size at which the miss rate is minimized for each of our applications. For most applications the optimal line size (with respect to miss rate) is less than or equal to 64 words.

The minimum *DTPR* is indicative of the amount of true sharing in a given application, since *DTPR* is always minimal when the cache line size is 1 word. The minimum

Table 4.1: Cache Line Size with Minimal Miss Rate.

Application	Minimal Miss Rate Cache Line Size
bsort	1024
gauss	128
kmerge	1024
matmult	16
mp3d	32
pgauss	16
plytrace	32
pmatmult	64
qsort	32
sorbyc	64
sorbyr	16384

miss rate depends on two independent characteristics of an application – processor locality (sharing) and spatial locality. When the cache line size is 1 word, the miss rate measures true sharing, and the extent to which this miss rate can be lowered using longer cache lines depends on the degree of spatial locality in the application. For most applications there is a tradeoff between lowering the miss rate with longer cache lines and a corresponding increase in *DTPR*. In the remainder of this section we discuss this tradeoff for specific applications.

Figures 4.1-4.3 show how a change in cache line size affects the miss rate and data transferred per reference for three applications: *pgauss*, *plytrace*, and *bsort*. We chose these three applications to illustrate various degrees of spatial locality, from poor locality (*pgauss*) to excellent locality (*bsort*). For each application we plot both the observed and ideal changes to *MissR* and *DTPR* as we increase the line size from 1 word to 128 words (i.e., 512 bytes). Our goal is to identify trends in how the miss rate and *DTPR* change with cache line size, and therefore we deliberately omit any scaling information on the y-axis.

Ideally, doubling the line size cuts the miss rate in half, and doesn't affect *DTPR*. Therefore, in figures 4.1-4.3 the ideal change in miss rate (relative to the miss rate for 1 word cache lines) is in proportion to the increase in line size, while the ideal *DTPR* (again relative to the *DTPR* for 1 word cache lines) remains constant. In reality, increasing the cache line size may not lower the miss rate proportionally and can affect *DTPR*; longer lines may fetch unused data and can also introduce false sharing. By plotting both the ideal and observed changes in *MissR* and *DTPR*, we can identify trends in how the miss rate and *DTPR* change with cache line size for a given application, and relate the memory reference patterns of the application to the trends.

It is evident from figures 4.1 and 4.2 that there is much more spatial locality present in *plytrace* than in *pgauss*. An increase in line size produces a greater decrease in *MissR* for *plytrace* than for *pgauss*. In fact, the changes in the observed miss rate are fairly

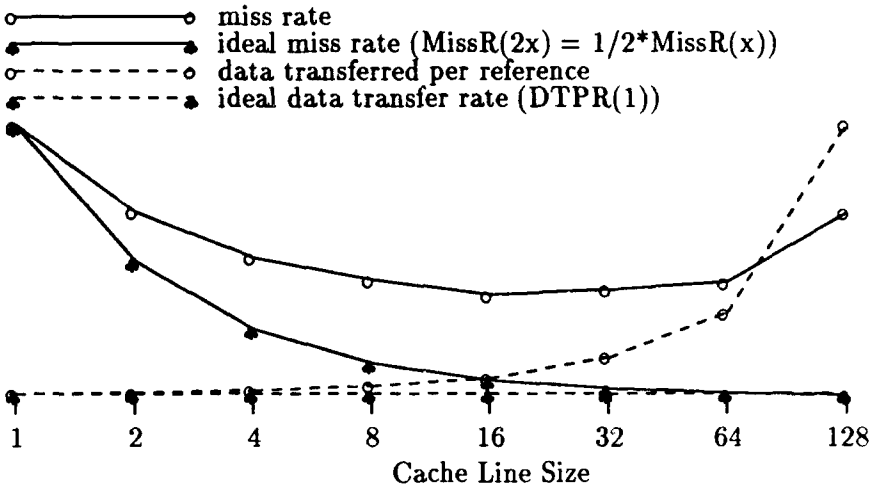


Figure 4.1: Relationship Between MissR and DTPR - *pgauss*

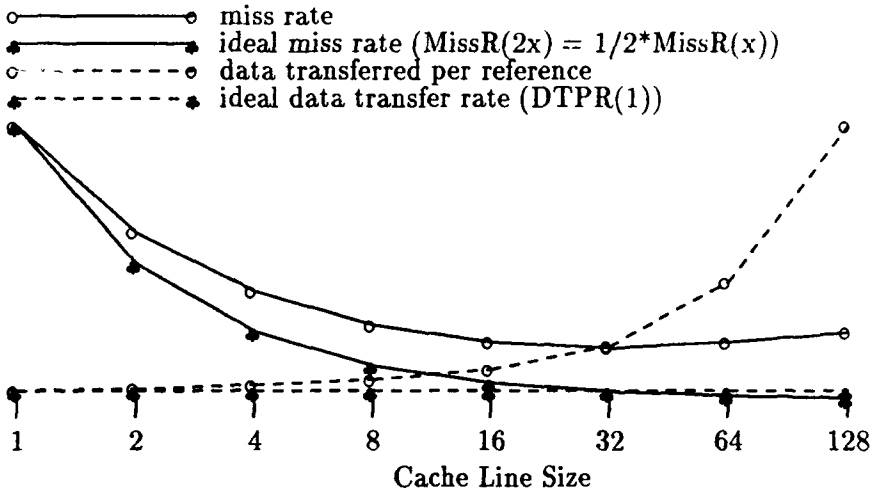


Figure 4.2: Relationship Between MissR and DTPR - *plytrace*

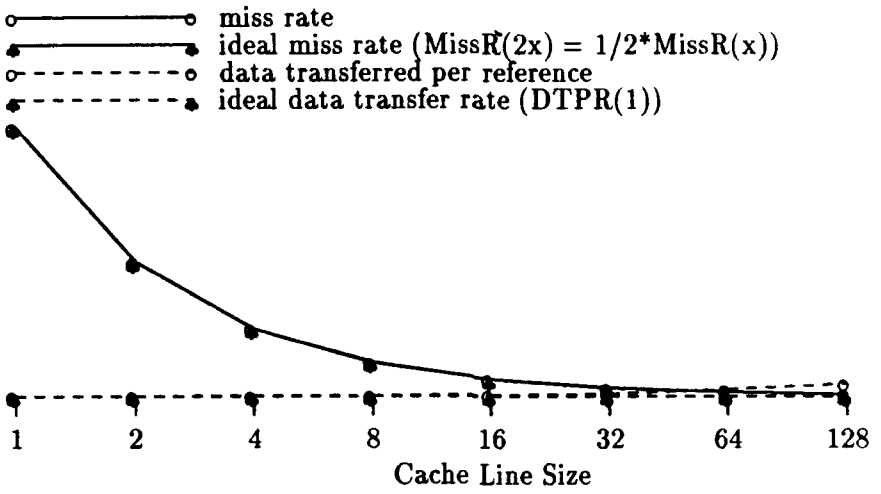


Figure 4.3: Relationship Between MissR and DTPR - *bsort*

close to the ideal for *plytrace* across most of the range of line sizes shown in the figure. For *pgauss*, false sharing causes the miss rate to rise when the cache line size grows above 16 words. The same effect occurs above 32 words for *plytrace*. With longer lines, the miss rate increases in proportion to the incidence of false sharing in the application.

For an application with extremely good spatial locality, such as *bsort* (figure 4.3), the miss rate decreases very quickly with an increase in line size. For this application, the observed miss rate for longer cache lines is essentially the ideal miss rate (i.e., doubling the cache line size cuts the miss rate in half). There is no false sharing effect in the range of line sizes we considered. Also, the data transferred per reference remains flat throughout the range of line sizes; that is, *DTPR* is equal to the ideal.

From these figures we can see that *MissR* and *DTPR* are almost always inversely related to line size. That is, the miss rate usually decreases with an increase in line size, while the data transferred per reference increases with the line size. If we ignore data transfers due to write-back and local misses, then *DTPR* for a given application is dependent on the miss rate according to the following equation:

$$DTPR(LineSize) = MissR(LineSize) * LineSize \tag{4.1}$$

That is, for a given application and cache line size (*LineSize*), the data transferred per reference is equal to the miss rate times the line size. Clearly then, spatial locality and false sharing influence both the miss rate and *DTPR*.

The rate at which *MissR* decreases with an increase in line size depends primarily on the spatial locality in an application. If the application exhibits good spatial locality, then an increase in line size produces a proportional decrease in *MissR*. If an application does not exhibit good spatial locality, then even if false sharing doesn't occur (as when doubling the line size from 1 word to 2 words in our case), increasing the line size may not produce a proportional decrease in *MissR*, since longer lines may fetch unneeded data.

Unlike the miss rate, *DTPR* routinely increases with an increase in cache line size. This is not surprising, since longer lines may fetch more data than is actually referenced, and unrelated data may share a cache line, and thereby introduce false sharing. A cache line size of 1 word fetches the minimum amount of data required for the correct execution of the program; the minimum miss rate cannot be generalized as easily, since it depends on the application.

4.2 The Effects of Bandwidth

In this section we consider how changes in the network bandwidth affect the *MCPR* for the programs in our application suite. We also investigate how the optimal (with respect to *MCPR*) cache line size for a given program depends on the network bandwidth. In these experiments we set the latency factor $F_{latency}$ to 50, corresponding to a round-trip time of 100 cache cycles per remote reference, which is characteristic of a scalable multiprocessor [39]. The impact of this assumption on our results is investigated in section 4.3.

Figures 4.4 to 4.9 plot *MCPR* for a given application as a function of the network bandwidth for a range of cache line sizes. We vary the bandwidth factor $F_{bandwidth}$ over the following range of values: $0, 1, 5, 10, 20$. Assuming 100 MHz processors and a four-byte wide network, these values for $F_{bandwidth}$ correspond to a machine whose bandwidth is infinite, 400 MB/sec, 80 MB/sec, 40 MB/sec, and 20 MB/sec respectively. (We consider a machine with infinite bandwidth so as to determine the extent to which our conclusions depend on the current state of network technology; future advances are more likely to increase the available bandwidth than to reduce the latency.)

Figure 4.4 illustrates the behavior of *bsort*. This application has very good spatial locality, and no false sharing, even for very long lines. Nearly all fetched data is referenced (recall the flat data transfer line from figure 4.3), so longer cache lines provide better performance regardless of the network characteristics. The curves representing different line sizes run in parallel in the graph, evidence that the *DTPR* remains flat.

We can see from figure 4.4 that doubling the cache line size offers diminishing returns for *MCPR*. For example, increasing the cache line size from 4 to 8 words substantially reduces *MCPR*, while an increase from 16 to 32 words offers much less improvement, and an increase from 64 to 128 words offers almost no improvement at all. Given the excellent spatial locality exhibited by this program, *DTPR* remains constant across these line sizes, and therefore the dominant factor in the *MCPR* is the miss rate. Doubling the cache line size cuts the miss rate in half for this program, so the improvement in *MCPR* decreases exponentially as we repeatedly double the line size. Thus, although we can expect longer cache lines to improve *MCPR*, the rate of improvement diminishes so rapidly with an increase in line size, that we quickly reach a point where further improvements in *MCPR* through longer cache lines are not worth pursuing (however, as will be shown in section 4.3 the relative utility of very long lines also depends on the latency).

Overall the best line size for this program (that is, the maximum line size) offers an *MCPR* very close to 1 for the high bandwidth machines, and around 1.6 for the

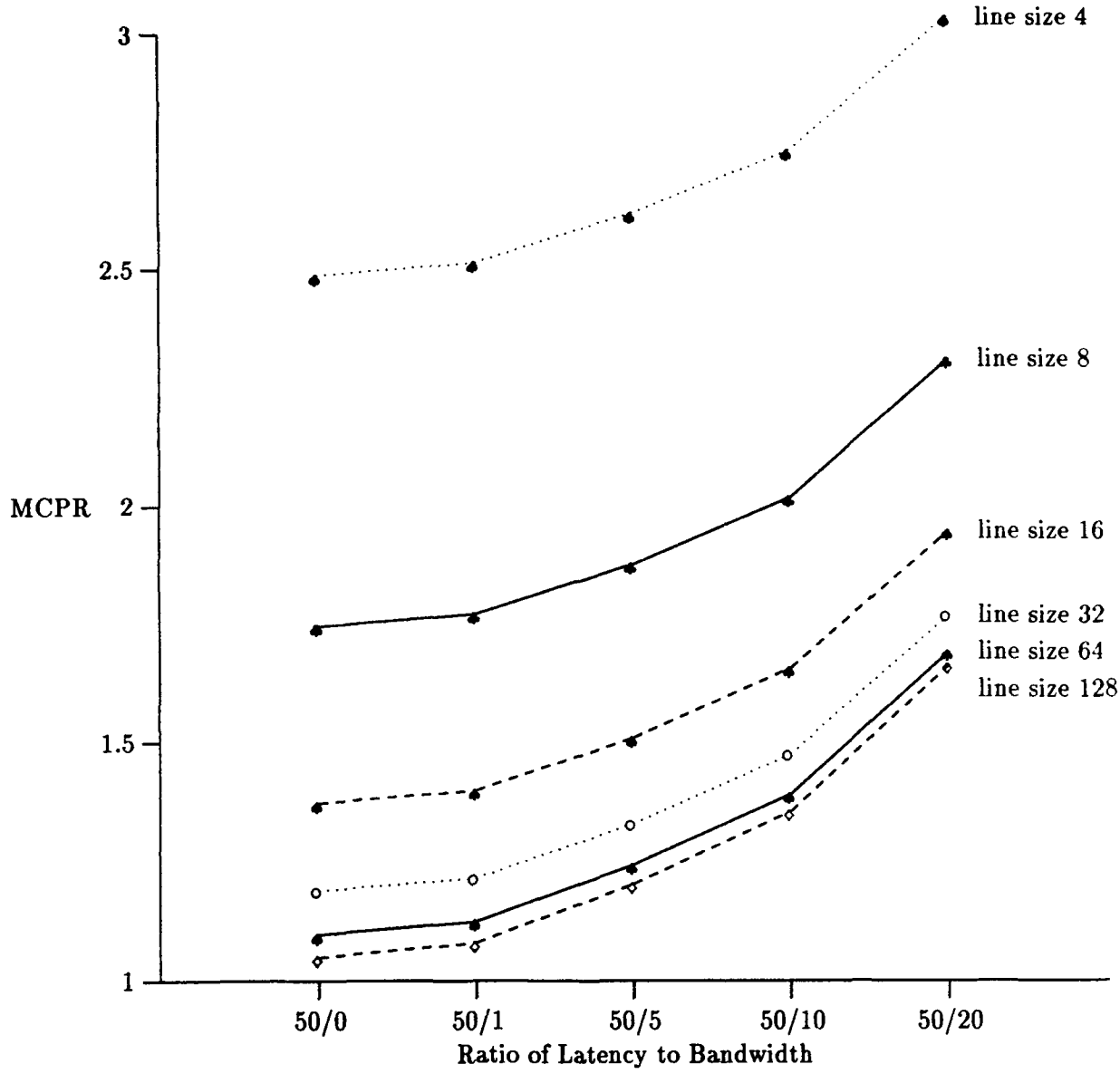


Figure 4.4: Mean Cost Per Reference as a Function of Bandwidth - *bsort*

low bandwidth machine. The low *MCPR* illustrates once again that this application is well-matched to shared-memory multiprocessors. Like *bsort*, the *MCPR* of *kmerge* is low for all machines, and close to 1 for the high bandwidth machines (figure not shown).

Sorbyr (in figure 4.5) displays characteristics similar to *bsort*, with two important differences. First, the longest cache line (128 words) apparently fetches more data than needed, which adversely affects performance on limited bandwidth machines (despite the lower miss rate provided by the long cache line). Thus, on the three highest bandwidth machines, the longest cache line (128 words) offers the lowest *MCPR* (due to the reduction in miss rate), while on the two machines with limited bandwidth, a cache line size of 64 words offers the lowest *MCPR*. The other difference between *sorbyr* and *bsort* is that the *MCPR* is even closer to 1 for *sorbyr*, which suggests that there is less sharing in *sorbyr* than in *bsort*.

Pmatmult (figure 4.6) exhibits worse spatial locality than either *sorbyr* or *bsort*. Although the curves for small cache lines run nearly in parallel, which indicates that little unused data is fetched, the performance of caches with longer lines deteriorates quickly with decreasing bandwidth. For the unlimited bandwidth machine, the best performance is delivered by a cache with line size of 64 words, which also minimizes the miss rate for this application. However, the best cache line size is 32 words on the other two high-bandwidth machines, and 16 words on the remaining low-bandwidth machines.

Gauss's performance as a function of a cache line size is similar to that of *pmatmult* except that the best *MCPR* is even lower, below 1.3 for all machines (figure not shown).

For most of the remaining applications, the longest line size shown in the figures is 32 words, because longer cache lines perform poorly. These applications prefer shorter lines and usually exhibit a much higher *MCPR* than the applications discussed above.

Matmult (figure 4.7) is an example of an application with low *MCPR*, which has more fine-grain sharing than *gauss* or *pmatmult*. The best line size is 16 words for the infinite-bandwidth machine and the high finite-bandwidth machine. It drops to 8 words for the three lower bandwidth machines. Analogous results (not shown here), but a higher *MCPR*, are displayed by *plytrace*.

Mp3d and *qsort* have comparable sensitivity to bandwidth; we will only discuss *mp3d* (figure 4.8). The infinite-bandwidth machine favors a line size of 32 words, which provides the minimum miss rate. The high-bandwidth machine also favors a 32-word cache line. The machine corresponding to a network with 80 MB/sec bandwidth favors a cache line with 16 words, and the other two machines with the lowest bandwidth prefer a line size of 8 words. This preference for short lines on low-bandwidth machines illustrates the limited spatial locality in this application. In addition, the lowest *MCPR* (on an infinite-bandwidth machine) is close to 10, which indicates a lot of sharing (i.e., poor processor locality). The best possible *MCPR* increases to over 20 on the lowest bandwidth machine. The range for *qsort* is better, between 4 and 7, but still represents a high *MCPR*, and the sensitivity to bandwidth measured as a relative change in *MCPR* is similar to that of *mp3d*. In both cases the best *MCPR* for the lowest bandwidth machine is about twice as high as for the infinite bandwidth machine. No other application displayed such extreme sensitivity to bandwidth.

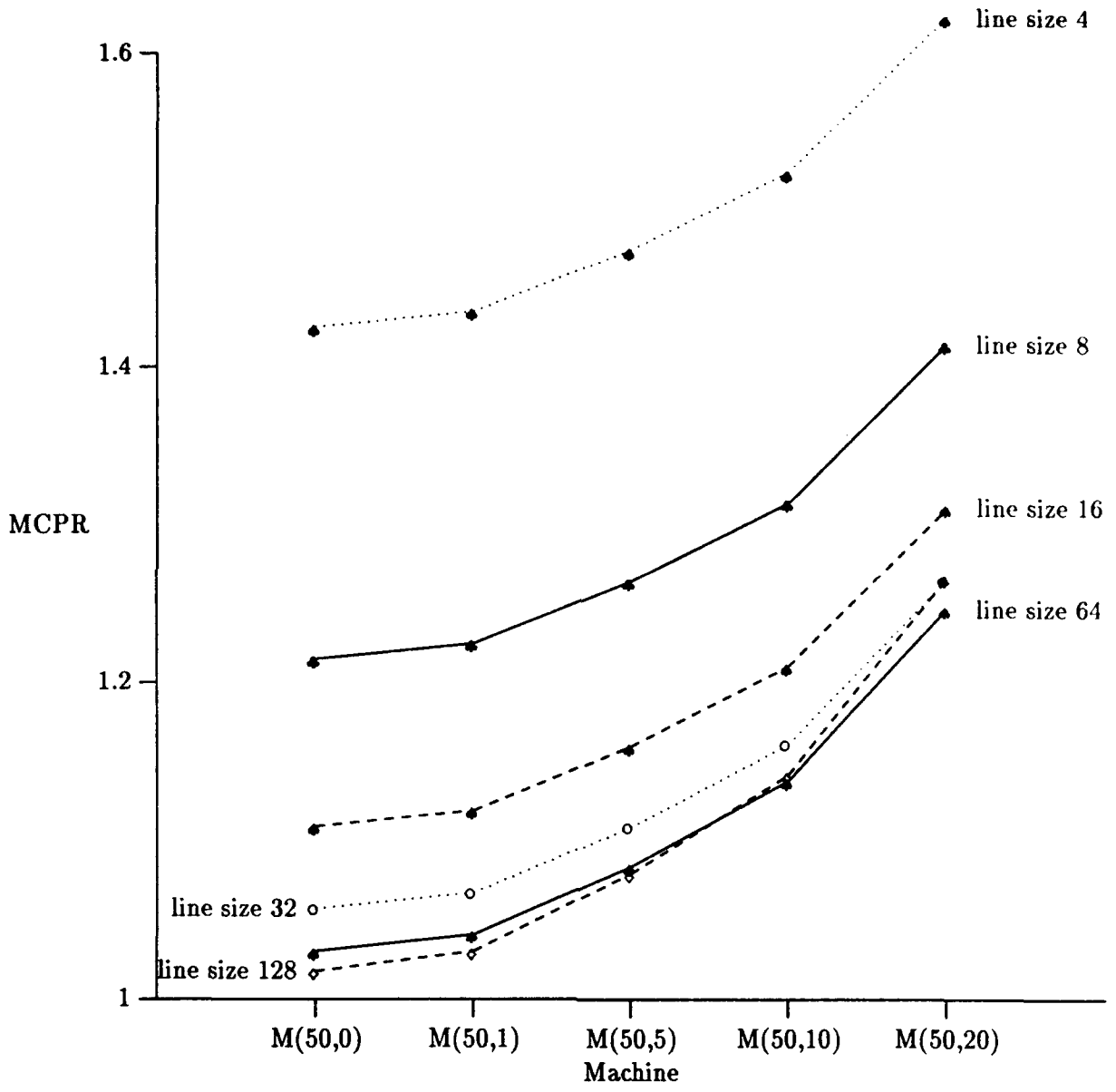


Figure 4.5: Mean Cost Per Reference as a Function of Bandwidth - *sorbyr*

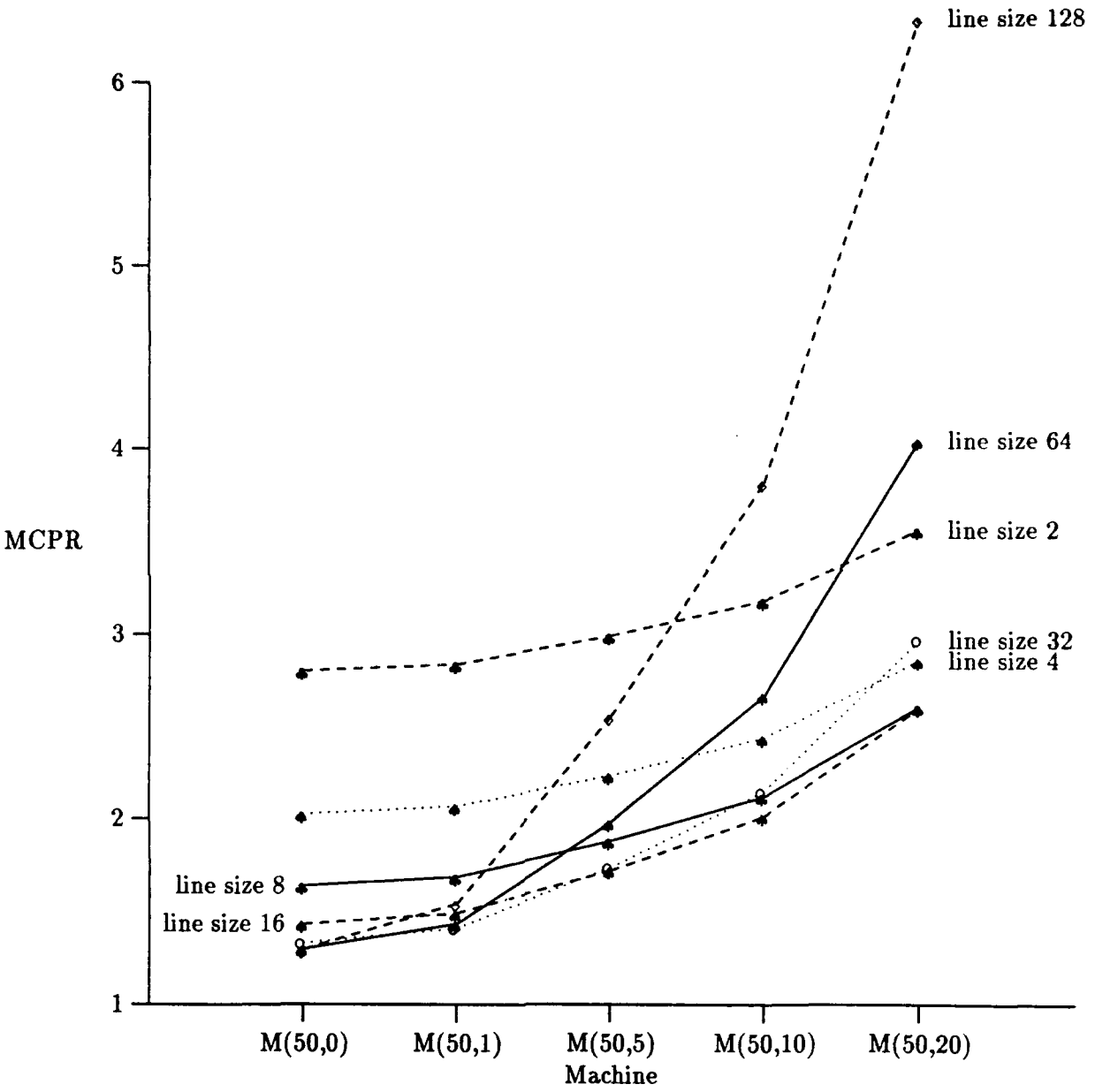


Figure 4.6: Mean Cost Per Reference as a Function of Bandwidth - *pmatmult*

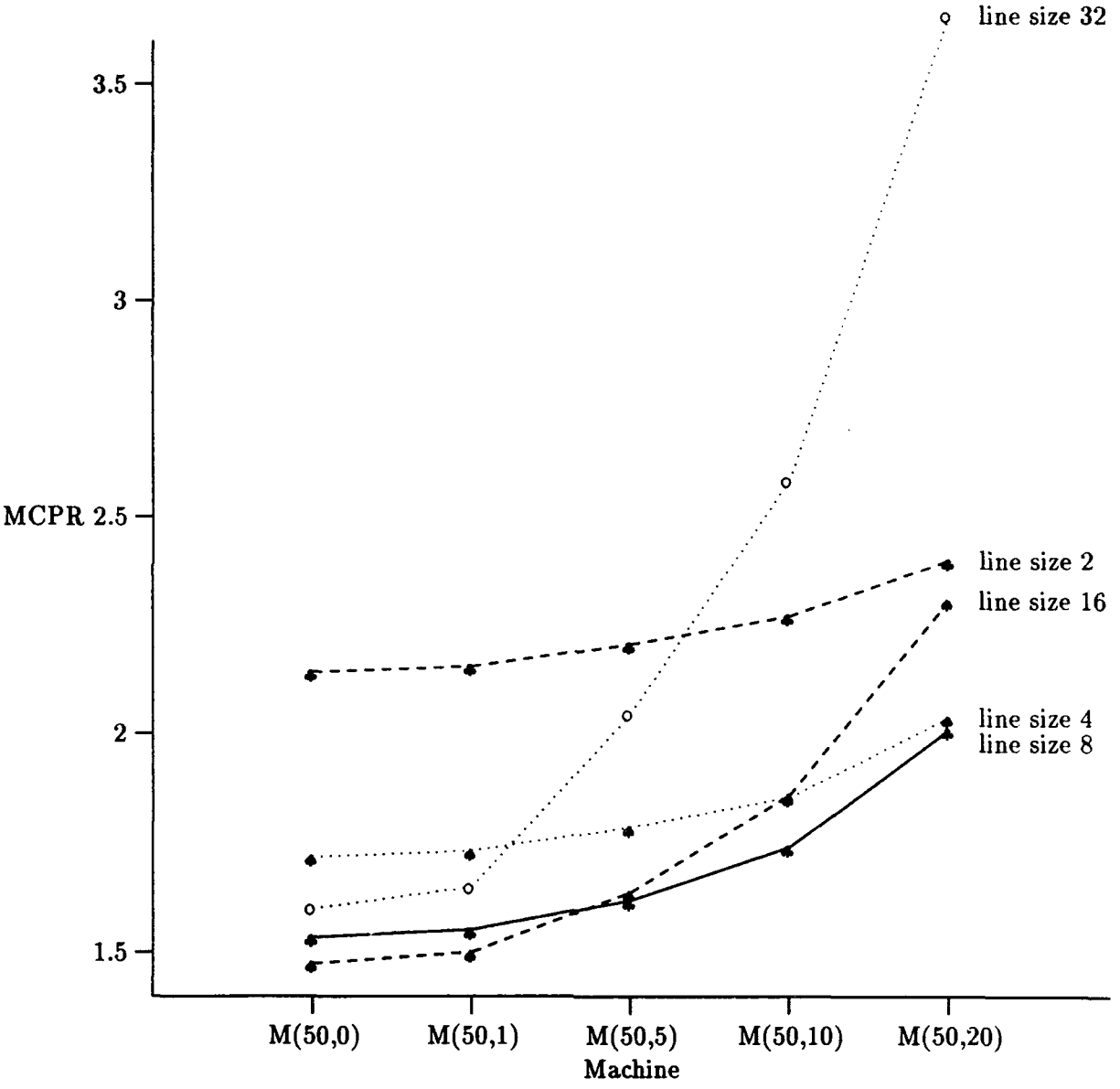


Figure 4.7: Mean Cost Per Reference as a Function of Bandwidth - *matmult*

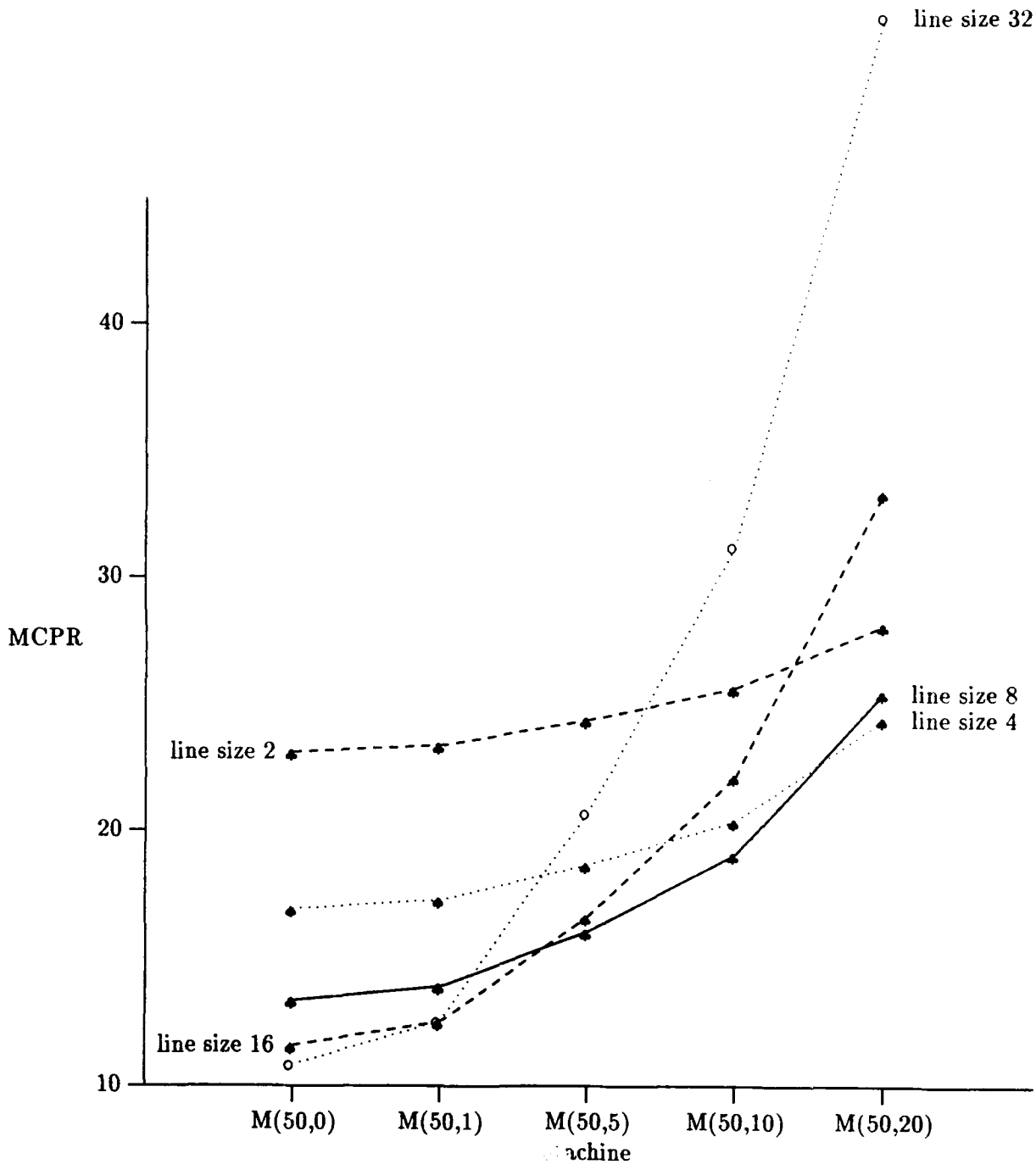


Figure 4.8: Mean Cost Per Reference as a Function of Bandwidth - *mp3d*

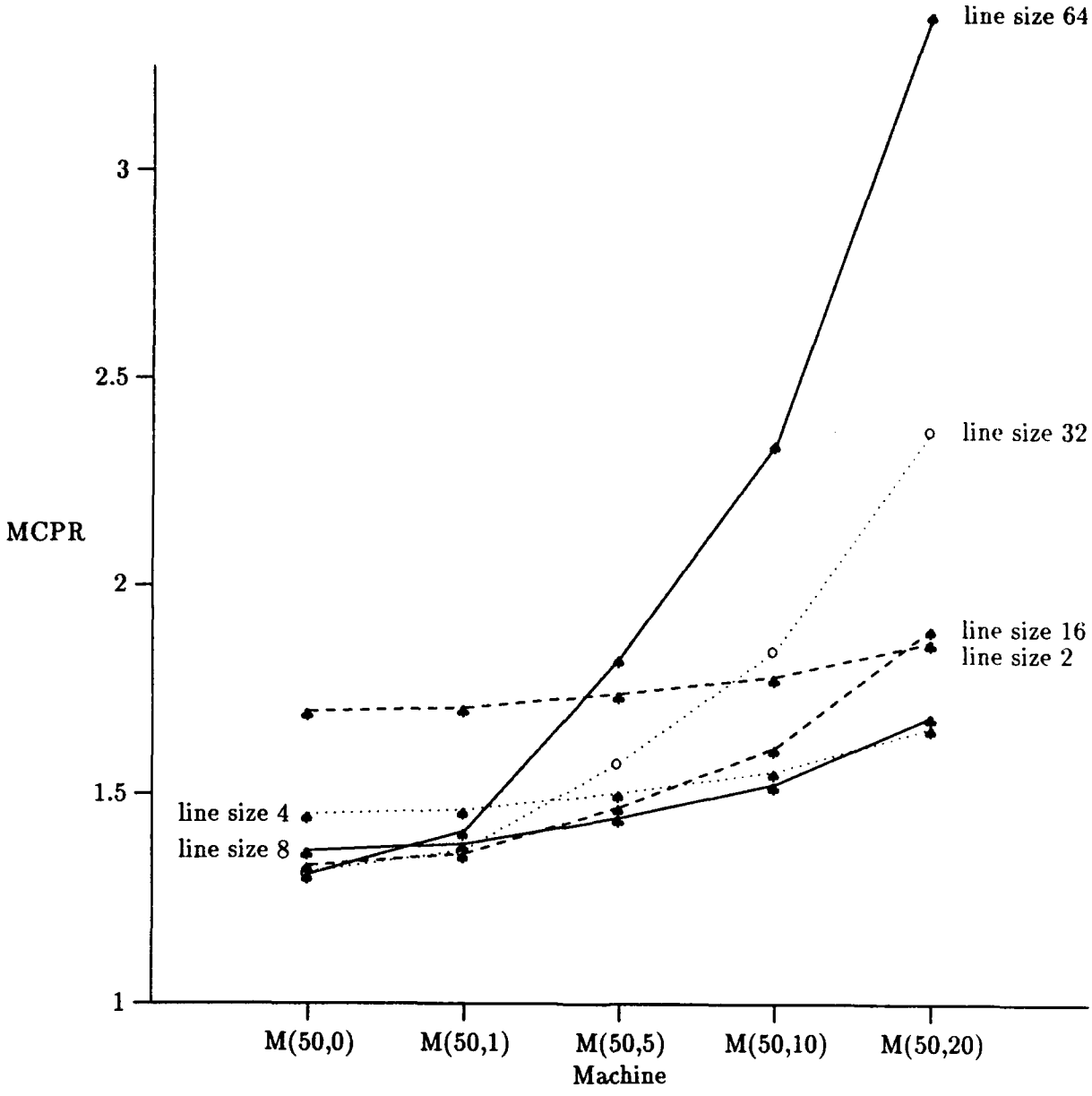


Figure 4.9: Mean Cost Per Reference as a Function of Bandwidth - *sorbyc*

Sorbyc is an interesting example of an application with little sharing, but the sharing that exists is fine-grain in nature. Figure 4.9 shows that the mean cost per reference (assuming the best possible line size) varies from 1.3 to 1.6 depending on the bandwidth. This fact seems to suggest that longer cache lines are preferred, as is the case on the high-bandwidth machines. However, for the two medium-bandwidth machines, the best cache line size is 8 words, and the limited-bandwidth machine prefers a line size of 4 words! Longer cache lines do not perform well for this program, due to a mismatch between the way data is referenced (by column) and the way the data is stored (by row). Thus, the mean cost per reference is higher when compared to *sorbyr*, but is low in absolute terms. The results for *pgauss* are similar, except that *pgauss* has a much higher *MCPR*. The best *MCPR* varies between 5 (on the infinite-bandwidth machine with a cache line size of 16 words) and 7 (the limited-bandwidth machine with a cache line size of 8 or 4 words).

We note here that good spatial locality does not guarantee a low *MCPR*. To achieve a low *MCPR*, we also need good processor locality (which amounts to a low degree of sharing), where data that is brought into a cache is used repeatedly by the associated processor. Both *sorbyr* and *gauss* have optimal *MCPR* below 1.3 on every machine, whereas *bsort*, which has the best spatial locality, has an *MCPR* greater than 1.6 in some cases. Moreover, programs with very similar spatial locality characteristics, such as *mp3d* and *sorbyc*, can have very different *MCPR* metrics due to different degrees of sharing.

In the remainder of this section we limit our attention to lines not longer than 64 words. Longer cache lines increase the likelihood of false sharing, and the benefits of increasing the cache line size soon reach a point of diminishing returns. Although long cache lines offer optimal performance for parallel programs with good spatial locality, such as *bsort* or *sorbyr*, the improvement in *MCPR* over a cache with 64-word lines is small (between 2% and 5% for *bsort* depending on the bandwidth, and only 1% for *sorbyr*). Since this result holds even on infinite-bandwidth machines, there is little argument in favor of increasing the cache line size beyond 64 words for these programs. On the other hand, for programs like *plytrace*, *sorbyc*, *pgauss* and *matmult*, cache lines longer than 64 words seriously degrade performance, and the amount of degradation depends on the bandwidth and the application sharing patterns. For *sorbyc*, which has both a low optimal *MCPR* and poor spatial locality, the *MCPR* increases by 60% on the lowest bandwidth machine when the cache line size increases from 64 words to 128 words. As the bandwidth increases, the performance penalty associated with the 128-word cache line starts to decrease, from 60% on the lowest bandwidth machine, to 41% on the 40 MB/sec machine, to 26% on the 80 MB/sec machine, and to 6% on the 400 MB/sec machine. The penalty practically disappears on the unlimited bandwidth machine. The same trend can be observed for other programs, although the penalty does not always go to zero with unlimited bandwidth. In particular, the penalty for *matmult* drops from 150% on the 20 MB/sec machine down to 17% on the unlimited bandwidth machine, and for *qsort* it drops from 133% to 21% when moving from the 20 MB/sec machine to the infinite bandwidth machine. Applications like *qsort* and *matmult* show that even with infinite bandwidth, the penalty of cache lines longer than 64 words can be substantial.

Table 4.2 illustrates how the best choice of cache line size depends on the ratio of $F_{latency}$ to $F_{bandwidth}$. Each entry in the table is the number of applications for which the given line size produced the best *MCPR* on the given machine. The blank entries mean that a particular line size was never the best for a given machine. The results presented in this table are valid for all machines described by the corresponding ratios of $F_{latency}$ to $F_{bandwidth}$, with the exception that the preference for a cache line size of 64 words indicates a preference for lines 64 words or longer.

Table 4.2: Best Fixed Line Size Histogram. An entry in this table indexed by cache line size and machine gives the number of applications for which that line size was the best fixed line size for that machine. Lines longer than 64 words are not considered.

$\frac{F_{lat}}{F_{band}}$	Cache Line Size						
	1	2	4	8	16	32	64
50/0					2	3	6
50/1					6	1	4
50/5				5	3		3
50/10			1	5	2		3
50/20			4	2	2		3

This table not only shows that different programs prefer different cache line sizes, but that the range of line sizes favored by our application suite changes with bandwidth (when latency is held constant). For the two highest bandwidth machines, the range is narrowest, and longer lines (16..64 words) are favored by all programs. When bandwidth is limited, the range of preferred sizes is much wider and the preferred line sizes are smaller. On the lowest bandwidth machine, most applications favor lines of 4 or 8 words, but not shorter. In general, we seek to minimize *DTPR* on low-bandwidth machines, thus shorter lines are preferable. We are more concerned with the miss rate on high-bandwidth machines, where longer lines are likely to perform best.

It is interesting to note how much performance can degrade for some programs if we restrict ourselves to one cache line size for each machine. No matter what line size we select, for each machine there is at least one application (never *mp3d*) that suffers between 11% and 36% increase in the *MCPR* (depending on the machine) compared to the lowest mean cost per reference for that application. This increase in MCPR, due to a mismatch between an application's sharing pattern and the cache line size, is based on the assumption that network latency is 50 cycles. In the next section we will consider how a change in latency affects the performance penalty associated with a mismatch between the application and the line size.

4.3 The Effects of Latency

Until now all our experiments have been performed on machines with $F_{latency}$ set to 50 cycles, i.e. a round trip latency of 100 cycles. Although we believe this latency is representative for a scalable multiprocessor, it is important to realize how our results would change with a change in latency. We are especially concerned here with latencies larger than 50 cycles, since they may be required for really large machines.

Increased latency makes reductions in the miss rate more important because there is now a higher cost associated with each miss. This in turn extends the range of preferred line sizes by including longer lines. Recall that in the previous section we decided to consider lines not longer than 64 words because lines 128 words long only improved performance for programs with excellent spatial locality, and by not more than 5%. However, this observation was made when the latency was 50 cycles. When we increase the latency, lines longer than 64 words offer higher improvements in *MCPR* (but only for programs with good spatial locality). Longer lines become more useful with higher latencies because the relatively small decrease in the miss rate becomes more important.

For example, in the case of *kmerge*, $Fixed(64)$ was worse by 4% than $Fixed(128)$ for $M(50,1)$. However, if we increase the latency to 100 cycles (machine $M(100,1)$), $Fixed(64)$ is 9% worse than $Fixed(128)$, and with latency 200 cycles (machine $M(200,1)$) it is worse by 17%. Clearly we cannot discard lines longer than 64 words for machines with latencies larger than 50 cycles. If we do not care about small differences in performance ($< 5\%$), then for machines with latency of 100 cycles, we can limit our attention to lines not longer than 128 words. For machines with latencies of 200 cycles, this limit is 256 words.

The scaling property can be used to illustrate that short cache lines become less useful with an increase in latency. Recall that in section 3.5 we showed that for any two caches their relative performance (measured in *MCPR*) on a given application P and a given machine $M(F_{latency}, F_{bandwidth})$ does not change if we scale both $F_{latency}$ and $F_{bandwidth}$ by the same constant $k > 0$. What changes is the magnitude of the difference in *MCPR* between the two caches, and for $k > 1$ this difference grows. Now consider what happens when we increase the latency of machine $M(50,20)$ by a factor of four to obtain machine $M(200,20)$. The ratio of $F_{latency}$ to $F_{bandwidth}$ in this new machine is 10, which is also the ratio of machine $M(50,5)$. From the scaling property, we know that the relative performance of the caches we consider is identical for both $M(50,5)$ and $M(200,20)$, and that the range of preferred line sizes for machine $M(200,20)$ is an upward extension of the range for machine $M(50,5)$. However, the range for $M(50,5)$ starts with a line size of 8 words, while the range for the original machine $M(50,20)$ starts with a line size of 4 words. In this case, increasing the latency by a factor of four, eliminated the need for cache lines of 4 words.

A similar argument illustrates that long cache lines become more useful with an increase in latency. For example, long cache lines offer greater performance benefits for $M(200,20)$ than for $M(50,5)$, because scaling by $k > 1$ magnifies differences in cache performance. As a result, the range of preferred line sizes for machine $M(200,20)$ includes 128 words, even though this line size did not offer significant improvements for

machine $M(50, 5)$.

Table 4.3: Ranges of Preferred Line Sizes. An entry in this table indexed by $F_{latency}$ and $F_{bandwidth}$ gives the range of preferred line sizes by our applications for machine $M(F_{latency}, F_{bandwidth})$. Longer lines are not included if the improvement in the *MCPR* is below 5%.

$F_{bandwidth}$	$F_{latency}$		
	50	100	200
0 (infinite)	16..64	16..128	16..256
1 (400 MB/sec)	16..64	16..128	16..256
5 (80 MB/sec)	8..64	16..128	16..256
10 (40 MB/sec)	4..64	8..128	8..128
20 (20 MB/sec)	4..32	8..64	8..128

Table 4.3 illustrates how the range of line sizes preferred by our applications changes with a change in latency and bandwidth. The upper limit for each range means that longer lines don't improve performance by more than 5% for any application on a given machine. The lower limit for each range means that there was at least one application for which a cache with that line size provided the best performance on a given machine.

We see that with an increase in latency, the range of preferred line sizes either extends or shifts upward. (Of course this change in the range does not continue forever since there is limited spatial locality in each application.) Increasing the bandwidth has greater effect on lower latency machines, where it shifts the range of preferred line sizes upward. With higher latencies, an initial increase in bandwidth also shifts the range upward, but soon the range of preferred line sizes becomes insensitive to increased bandwidth for these machines.

Not unexpectedly, an increase in latency exacerbates the performance effects of a mismatch between the cache line size and an application's sharing pattern. For example, consider two groups of machines: (1) the $M(100, *)$ group, which consists of the five machines with $F_{bandwidth}$ equal to 0, 1, 5, 10, and 20, and latency equal to 100, and (2) the $M(200, *)$ group, which consists of five machines with the same range for $F_{bandwidth}$, but with latency equal to 200. If we consider all cache line sizes between 4 and 256 words (in powers of two), and attempt to select a single line size for each machine, we find that for the $M(100, *)$ group, there is always at least one application that suffers an increase in *MCPR* between 26% and 44% (depending on the bandwidth) compared to the performance provided by the best fixed line size cache for that application. For the $M(200, *)$ group, this increase in *MCPR* is between 37% and 68% (again, depending on the bandwidth). In general, the performance degradation in *MCPR* due to a mismatch between the cache line size and the application increases with increased latency, and decreases with increased bandwidth.

4.4 Summary

In this chapter we evaluated the performance of fixed line size caches. First, we examined the relationship between the miss rate and the amount of data transferred per reference for our application suite. Next we considered how increases in network latency or bandwidth impact the performance of fixed line size coherent caches in scalable multiprocessors. Using application reference traces that cover a wide range of sharing behavior, and a multiprocessor simulator that allows variations in network costs, we examined the relationship between application sharing behavior, cache line size, network latency and bandwidth, and performance (as captured by the mean cost per reference).

Our results suggest that, regardless of the available bandwidth or latency, applications with good spatial locality favor long cache lines, and for these applications the relative benefits of longer cache lines increase with the bandwidth and latency. For those applications with poor spatial locality, the best choice of cache line size depends on the ratio of $F_{latency}$ to $F_{bandwidth}$, and for these applications, the performance penalty induced by long cache lines increases as the ratio decreases.

Our results also suggest that unlimited bandwidth (above 400 MB/sec) does not significantly improve the mean cost per reference for most applications, and in particular, does not produce a low mean cost per reference for applications with fine-grain sharing or poor spatial locality. When the latency is 50 cycles there is not much difference between the best *MCPR* on the infinite-bandwidth machine and the best *MCPR* on the next highest bandwidth machine for all applications except *mp3d* and *qsort*. Thus, an increase in bandwidth beyond 400 MB/sec is not likely to reduce *MCPR* substantially. The impact of high bandwidth is even less noticeable on the machines with higher latencies.

With respect to the range of preferred line sizes, our results indicate that there is little benefit to be gained from using cache lines shorter than 4 words for scalable multiprocessors. The utility of short lines decreases as bandwidth and latency increase, and our simulations suggest that cache lines less than 4 words long are unnecessary on current machines.

While it is no surprise that very short cache lines are unnecessary in the absence of synchronization references, one might expect long lines to be of substantial benefit to programs with excellent spatial locality. We found however that the incremental performance improvements of using long cache lines diminish rapidly, and there is always a line size beyond which these improvements are negligible, even when an application has enough spatial locality to make use of long cache lines. This upper limit on line size depends on the network latency of the host multiprocessor, and for latency equal to 50 cycles, it is equal to 64 words. Additionally, longer cache lines increase the likelihood of false sharing. Thus, the benefits of even longer lines do not justify the increase in invalidations suffered by programs that favor shorter lines.

With respect to the range of preferred line sizes favored by programs in our application suite we also found that it shifts upward with an increase in latency. This range moves upward with an initial increase in bandwidth, but at some point the higher

bandwidth does not affect the range. The higher the latency, the earlier increases in the bandwidth stop influencing the range of preferred line sizes.

To estimate the effect of a mismatch between the cache line size and an application's sharing pattern we investigated how much the *MCPR* increases from the lowest per-application *MCPR* if we select one cache line size for all applications. We note that this increase gives a lower bound on the penalty associated with a mismatch between the line size and the granularity of sharing exhibited by a program. For at least one application this increase was between 11% and 36% (depending on the bandwidth) for $M(50,*)$ machines, between 26% and 44% for $M(100,*)$ machines, and between 37% and 68% for $M(200,*)$ machines. Generally, the penalty decreases with increased bandwidth, but we can see that even for infinite bandwidth machines it remains substantial. Also, there is a strong positive correlation between network latency and this penalty.

As we can see, the penalty associated with a mismatch between the line size and an application's sharing pattern is substantial and will likely be a serious problem in future scalable machines. Therefore we need an effective solution that eliminates this penalty, which provides the motivation for the adjustable block size coherent caches introduced in the next chapter.

5 Adjustable Block Size Caches

This chapter describes the idea and details of a new cache organization called adjustable block size caches. Section 5.1 gives an overview of the new cache organization. Section 5.2 describes the architectural extensions to the multiprocessor introduced in section 3.1 needed for implementation of adjustable caches. The details of adjustable cache implementations are given in section 5.3. We discuss the integration of dynamic block adjustment with the coherency protocol in section 5.4. A summary of this chapter is given in section 5.5.

5.1 Overview

Our new coherent cache organization adjusts the data block size to match the granularity of sharing observed during program execution. It uses a power-of-two buddy scheme to split and merge data blocks based on recent access patterns. Under this scheme, a data block is split in two when a processor references only one half of the data block. Two adjacent data blocks of the same size are merged when both halves of both blocks are referenced by a single processor. Initially all blocks are of equal size, called the initial block size.

More than one copy of a given data block may exist in caches associated with different processors, but all such copies are read-only. Only one writable copy of a data block may exist.

Logically each copy of a data block has a number of fields associated with it. In addition to the data field and address tag field the following fields are associated with each copy of a data block:

- *size* - Indicates the size of the data block. The size may be any integer power of two from *MinBlockSize* to *MaxBlockSize*.
- *LU* - Reference bits for the lower and upper half of the data block. Each time data is referenced in one half of the data block, the corresponding reference bit is set.
- *split-merge counter* - Records the number of processors that accessed only the lower or upper half of the data block while it was resident in their caches. The

counter is incremented if only one half of the data block was accessed, and is decremented if both halves of the data block were accessed.

The mapping between data blocks and cache lines depends on the implementation of adjustable caches, which is discussed in section 5.2.1.

A data block may be split or merged when it has been modified by one processor and is then requested by another processor. Only one copy of this data block exists in such cases. [1] The decision to split a data block is based on the value of the split-merge counter associated with it. The counter is updated according to the value of the reference bits when the data block is requested by another processor; the value of the split-merge counter is transferred to the requesting processor with the contents of the block. A positive counter suggests that processors do not on average access both halves of the data block, and therefore the data block should be split in two. A negative split-merge counter suggests that processors do access both halves, and therefore the data block should not be split.

The decision to merge two adjacent data blocks is based on the value of their respective split-merge counters. Merging is encouraged when adjacent data blocks of the same size both have negative counters.

Within this basic organization, there are several parameters that can affect how quickly and accurately the data block size converges to the desired value. The maximum block size must be large enough to exploit spatial locality. The minimum block size must be small enough to avoid false sharing, The initial block size must be close to the average block size, if data blocks are to converge to an appropriate size quickly.

The accuracy of the reference information and the way it is used also affects how quickly the desired block size is reached. Rather than allow the short-term reference pattern of a single processor to determine when to split or merge cache blocks, we define separate split and merge threshold values that must be exceeded before a split or merge occurs. High thresholds can discourage excessive splitting or merging, but the thresholds must not be so high as to prevent splitting or merging. When the split (merge) threshold is N, a data block must have been modified and then requested by another processor N times before the block can be split (merged).

An instance of the adjustable block size cache is denoted by

$$Vblock(MinBlockS, MaxBlockS, InitBlockS, (SplitThr, MergeThr))$$

where $MinBlockS$ is the minimum block size, $MaxBlockS$ is the maximum block size, $InitBlockS$ is the initial block size, $SplitThr$ is the split threshold and $MergeThr$ is the merge threshold.

Regardless of the specific values chosen for these parameters, the main advantage of this cache organization is that the size of data blocks varies depending on references to the data. When appropriate, the largest possible block size is used, minimizing the

[1] The block size adjustment is restricted to modified blocks because there is only one copy to consider. It is possible to adjust the block size of read-shared blocks, but it would incur the cost of reaching a global consensus on the new block size.

number of bus or network transactions needed to retrieve the data. False sharing causes a data block to be split, reducing the number of subsequent invalidations. We would expect this adaptive scheme to produce fewer invalidations than a cache with a large, fixed block size, and require fewer bus or network transactions than a cache with a small, fixed block size.

5.2 Implementation in a Scalable Multiprocessor

We will now illustrate the implementation of an adjustable block size cache in the scalable, shared-memory multiprocessor described in section 3.1.

5.2.1 Processor Caches

The challenge in implementing an adjustable block size processor cache is to map each data block's control and data fields into cache lines. Our goal is to minimize the cache cycle time, provide good cache utilization, and avoid excessive complexity.

We assume that all processor caches are fully associative, and all cache lines are of the same size. We propose two alternative implementations of an adjustable block size cache. These implementations differ in the size of a cache line. The *max-block* implementation assumes that the size of the data part in each cache line is equal to the maximum block size allowed by the adjustable block size cache. The *min-block* implementation assumes that the size of the data part in each cache line is equal to the minimum block size. In both implementations each cache line contains space for a tag for a block of the minimum block size. Additionally, each cache line includes control fields, like valid and modify bits, block size bits, block state bits, two reference bits and a split-merge counter.

In the *max-block* implementation, a data block occupies exactly one cache line. If a block is smaller than the maximum block size, some space in the cache line is wasted. Under this scheme, the effective line size of the cache is equal to the average data block size, which could be much smaller than the actual line size. On the other hand, splitting and merging is easy because all data for a given block is in just one cache line.

In the *min-block* implementation a copy of a data block larger than the minimum block size occupies more than one cache line. Such a block is represented by a head cache line, which contains the first part of the data block, and tail cache lines, which contain the remainder of the data block. Each cache line tag contains a valid tag corresponding to the part of the data block kept in the line. The head line for a block is different from the tail lines because it keeps the actual size of the block, as well as other control fields, like reference bits and the *split-merge* counter. These control fields are unused in tail lines. We need one bit per cache line to indicate the head/tail status of each line.

To split and merge blocks in this organization we manipulate only the head/tail status of cache lines and associated control fields, without moving any data. Fetching a block from the cache is now more difficult because blocks may span cache lines and we do not know the size of a block. To fetch the block at a given address, we first fetch the

head line. Next we extract the size of the block from the size bits of the head line and fetch subsequent tail lines (if any). We note that fetching or invalidation of the whole block is needed only on the coherency transaction. Therefore fetching the tail lines can be overlapped with the network transmission. On a cache hit we do not need access to the whole block; all we need to do is update the reference bits and fetch the referenced data, which can be easily done in parallel.

An implementation with cache lines of the maximum block size sometimes wastes cache line data space, whereas the alternative implementation with lines of the minimum block size uses many more cache lines, and many more tag and control parts of cache lines, most of which are wasted for blocks larger than the minimum block size. The tradeoffs in cache space introduced by these two implementations are quantified in section 6.4.

In general, we would expect an adjustable block size cache to be used as a second-level cache. Doing so would shield the processor from the effects on cycle time caused by complications in the logic of an adjustable block size cache. The first-level cache could use very short lines (say, one word) and be direct-mapped. None of the fields introduced above would have to be stored in the first-level cache. The reference bits for a block in the second-level cache would be updated on the first reference to a sub-block, when the sub-block is brought into the first-level cache. Subsequent references to the sub-block would not require accesses to the second-level cache, as long as the data remains in the first-level cache.

In the remainder of this work, we will ignore the details of interaction between the first and second-level caches, focusing instead on the role of the adjustable block size cache in coherency transactions.

5.2.2 Tag Caches

Every memory module in the simulated machine organization has a tag cache [43], which is used to implement a distributed directory. A tag cache is fully associative and indexed by block address. Each tag cache line maintains information about the nodes that currently have a copy of the data block indexing the tag cache line.

As originally described in [43], a tag cache actually consists of two subcaches - one large cache for short cache lines, and a smaller cache for long cache lines. A short cache line contains a few pointers to processors that currently cache the data block. Only a few processors can have a copy of a block that has been assigned to a short tag cache line. If the number of processors sharing a block of data exceeds the number of pointers in a short cache line, then a long cache line is used. Long cache lines keep track of the copies of the data with a bit vector instead of pointers. The bit vector has an entry for each processor in the system, so a data block associated with it can be cached by every processor in the system.

When a block is first referenced it is allocated a short line. When the number of processors caching the block exceeds the number of pointers in a short line, the short line is freed, and a long line is allocated. When there are no free tag lines, one tag line is freed. Tag lines within each tag cache are freed on an LRU basis.

Tag caches make use of the observation that only a few blocks are shared by more than a few processors. Therefore, there is no need to maintain a full-length bit vector for most of the blocks. Another important observation is that tag caches need only maintain directory entries for blocks currently cached in processor caches. Hence the size of the directory is limited by the sum of the sizes of the processor caches, and not by the total size of the memory modules.

Tag caches are useful not only for their memory-efficient representation of directories, but also because they admit an implementation of the adjustable cache organization (as discussed in section 5.1). Since the potential cost of limited pointer space in tag caches is not a focus of our work, we assume that all tag cache lines are long. Even with long tag lines, tag caches consume much less space than the standard directory organization.

5.3 Adjusting the Block Size

In addition to maintaining data coherency, the coherency protocol described in section 3.2 must be extended to distribute reference information, and to choose when to split or merge cache blocks.

The reference bits associated with a copy of a data block record reference information on a per-processor basis. These bits are used to update the split-merge counter, which records reference information across processors. The split-merge counter is updated as follows:

- When a requesting node receives data directly from the home node, it clears its split-merge counter.
- When a data block changes owner, the previous owner updates its split-merge counter based on the current values of its reference bits, and includes the new value of its counter with the data. The new owner initializes its counter to this value. If the change in ownership results in a split or merge, the new owner initializes its split-merge counter to zero.
- When a node receives invalidation acknowledgements from other nodes containing copies of a data block, the acknowledgements include the split-merge counters for each copy. The requesting node adds these values to its split-merge counter.

The decision to split a data block is made by the owner on a transition in ownership. When an owner receives a request forwarded from the home node, it updates its split-merge counter and compares the new value to the split threshold. If the split-merge counter exceeds the split threshold, a split occurs. The owner provides the requesting node with the half of the original block containing the requested offset within the block. The original owner maintains ownership of half the block (clearing the split-merge counter for the block), and transfers ownership of the other half (containing the requested offset). The home node updates its tag cache and memory (if necessary) when notified of the split by the original owner, as part of the sharing-writeback message or

ownership-change message. Thus, no additional messages are required to implement a split, and the amount of data transferred is actually less than in the case where a split does not occur.

Two adjacent data blocks of equal size can be merged to form a single block at two different points:

1. If we load the tag for the longest possible block on a tag cache miss, we implicitly merge blocks after the block tags are evicted from the tag cache.

2. A merge can be ordered by the owner when a block in the *modified* state is requested by another node. The following conditions should be met when a merge is ordered by the owner: (a) both blocks are owned by the same node, (b) they are buddies in the power-of-two buddy scheme, and (c) the split-merge counters of both blocks exceed the merge threshold.

The owner invalidates the processor cache lines containing the two subblocks being merged on a write request, and changes the state of the blocks to *read-shared* on a read request. The owner also sends a *reply* message containing the merged blocks to the requesting node. Merging does not require extra messages, since the home node is notified as part of the sharing-writeback message or ownership-change message, but the amount of data transferred is increased because both the requested block and its neighbor are sent to the requesting node.

If a merge is attempted but cannot be performed because of the state of the buddy block we call it a failed merge.

In our simulations we charge 2 cache cycles for a split, 4 for a merge, and 1 for a failed merge. This additional cost is reflected in *MCPR*, measured in cache cycles of the processor cache. We make merge operations more expensive than splits because merge is more complicated, i.e. more conditions need to be checked and, in case of a successful merge, more state needs to be updated.

5.4 Integration with the Coherency Protocol

The directory entry for a given data block attempts to maintain up-to-date information associated with the block including its coherency state and the distribution of its copies. Unfortunately, multiple coherency transactions related to one data block can be simultaneously in progress. In such cases the information maintained by the directory may become out of date.

In the original DASH protocol, both the block state and its distribution may change while the directory keeps old information. For example, the owner of a *modified* block may decide to evict it from the cache in order to reclaim the cache line. In such a case the owner sends an eviction writeback message to the directory, so it will be eventually updated. In the meantime, there may be another request issued for the data block, which would result in the request being forwarded to the old owner. When this request

reaches the old owner, it replies with a negative acknowledgment (NAK), which will cause the request to be re-issued later, upon reception of a NAK message. Interested readers may find details of this solution in [38].

Extending the DASH protocol with dynamic block adjustment requires that the directory keep track of the block size, in addition to any previously maintained information. Moreover, dynamic block adjustment may result in a situation where the block size stored in a directory entry is not the actual block size. This may happen when the owner of a *modified* block has split it in two or has merged it with its buddy block, and the sharing-writeback message or ownership-change message conveying the information about this adjustment has not yet reached the directory.

To deal with such cases, we propose to lock the directory entry each time a request is forwarded to the owner. When a read request is forwarded, the entry remains locked until a sharing-writeback message is received by the home node. When a write request is forwarded, the entry remains locked until the home node receives an ownership-change message. In both cases the directory entry is unlocked when the writeback message generated by block eviction is received by the home node. While the directory entry remains locked, any request for a data block represented by this entry is rejected by sending a NAK message to the originating node.

This locking scheme solves the out-of-date block size problem in cases of splits because the associated entry remains locked while a split is in progress. Merges are more complicated, because we can have two requests forwarded to two buddy blocks, both of which are locked. The first request reaching the owner (which owns both blocks) causes the merge to occur, followed by a sharing-writeback or ownership-change message. When the second request reaches the owner, it no longer has the requested data, so it must answer with a NAK message, as in the original DASH protocol after eviction. Upon receiving a sharing-writeback or ownership-change message containing merge information, the home node unlocks and deletes both directory entries corresponding to the two merged sub-blocks, and creates one entry representing a newly merged block. This new entry is unlocked, as the home node knows, due to the merge information received, that the second request has not been satisfied.

It is easy to verify this extended protocol is correct, since each of the possible actions in the extended protocol corresponds to a legitimate scenario in the DASH protocol. Each locked entry eventually will be unlocked, because the owner must reply with one of the following messages affecting the locked entry: sharing-writeback, ownership-change, eviction writeback. Splits do not introduce any new cases in the protocol since (due to locking) requests for data in a block being split are postponed (using NAK) until the split is completed. Merging introduces no new cases as long as no requests are issued for a block after it has merged with its buddy but before the directory has been notified. There is one special case however; if a request to the buddy block is issued while merging is in progress, the request cannot be postponed because the buddy block is not locked. (Anticipatory locking of the buddy block is one possible solution, but it would adversely affect the performance of the protocol.) This case is analogous to a situation that can arise in the DASH protocol: (1) the home node receives and forwards two separate requests for two data blocks owned by the same owner, (2) the second

request is to the buddy block of the first request, (3) the owner happens to evict the block corresponding to the second request after the first request has been received, but before the second request arrives. The only difference between these two cases is that in our protocol, the home node would receive merging information as part of a sharing-writeback or ownership-change message, while in the DASH protocol the home node would receive a sharing-writeback or ownership-change message followed by an eviction writeback message. Since we can translate protocol actions on merge and split operations into legitimate DASH operations, the extended protocol functions correctly if the DASH protocol is correct.

5.5 Summary

In this chapter we introduced a new cache organization with an adjustable block size, which is intended specifically for shared-memory multiprocessors. In this organization the size of the coherency unit (block size) is not constant but may vary depending on the particular block and point in time. The recent reference pattern is used to adjust the block size dynamically by splitting blocks to minimize false sharing and by merging blocks to maximize the prefetching effect of large blocks.

We described two alternative implementations of processor caches for our new cache organization. In one scheme each cache line is large enough to accommodate a block of the maximum allowed size; the other scheme assumes that blocks may span multiple short cache lines.

Finally we discussed the details of adjusting the block size and how it is integrated with a directory-based coherency protocol.

We expect adjustable caches to provide good performance for a wide range of sharing behaviors exhibited by parallel applications. The most important question in the evaluation of an adjustable cache is to determine how well it performs compared with traditional fixed line size caches. We need to quantify this comparison not only with regard to sharing behavior of parallel programs but also with regard to the characterization of the host multiprocessor, especially the network latency and bandwidth. It is also interesting to see if a prefetching cache offers performance competitive with our new cache organization. The cache comparison study should be performed for both network-independent metrics, like the miss rate and *DTPR*, and machine-specific metrics, like the mean cost per reference. This complete approach should help us understand the behavior of adjustable caches as well as offer specific performance numbers. We also need to quantify the overhead in cache space introduced by adjustable caches. Another important issue is to identify how to select an instance of the adjustable cache for a particular machine and a representative set of applications. All of these issues will be addressed in the following chapters.

6 Evaluation of Adjustable Block Size Caches

This chapter contains our evaluation of the adjustable block size cache organization, which we compare with more traditional fixed line size caches with various degrees of multi-line prefetching. In section 6.1 we describe the instances of cache organizations to be included in our comparison study; the selection of adjustable caches is motivated by the results of chapter 4 on the ranges of line sizes preferred by our applications. In section 6.2 we compare the performance of the selected caches using network-independent metrics, like the miss rate, *DTPR*, and cache size. We examine the impact of dynamic block adjustment on the mean cost per reference in section 6.3. We compare the amount of adjustable cache space required for conflict-free execution of our applications with the space requirements of fixed line size caches in section 6.4. We summarize our evaluation of adjustable block size caches in section 6.5.

6.1 Cache Instances

In this section we define the instances of cache organizations used in the comparison study. We focus on the $M(50,*)$ machines introduced in section 4.2. The effects of latencies higher than 50 cycles are discussed in section 7.5.

For the fixed line size cache organization we select four caches with line sizes between 4 and 64 words in powers of two. These upper and lower bounds follow from the discussion in section 4.2. We saw that under no circumstances is a fixed line size less than 4 words optimal for any of our applications (see table 4.2). When latency is 50 cycles, lines longer than 64 words resulted in serious performance degradation for most applications, and the improvements in *MCPR* were negligible for the others.

We will use three instances of adjustable block size caches in our comparison. Our choices are motivated by the wide range of architectures we want to cover, but for any specific machine we will advocate one particular instance of the adjustable cache. The three instances selected are:

- $Vblock(16,64,64,(1,1))$ (or $Vblock(16,64,64)$)
- $Vblock(8,64,64,(1,1))$ (or $Vblock(8,64,64)$)
- $Vblock(4,64,16,(1,1))$ (or $Vblock(4,64,16)$)

All these instances use the same values for split and merge thresholds, therefore we will omit these thresholds in our short form notation.

The first instance, *Vblock*(16,64,64), is intended for high bandwidth machines because it encourages the use of longer lines and prevents the use of short and medium lines, since the minimum block size is 16 words. The range 16..64 includes the optimal line sizes for unlimited bandwidth machines and high bandwidth machines (see table 4.2).

The second instance, *Vblock*(8,64,64), extends the range to include blocks of size 8 words. This block range allows this instance to cover all optimal line sizes for the medium bandwidth machine (with $F_{bandwidth}$ equal to 5, which corresponds to 80 MB/sec in our model).

The third instance, *Vblock*(4,64,16), extends the range of block sizes even further to include blocks of 4 words. Also, the initial block size is now 16 words. This cache instance is better suited to the two low bandwidth machines because most of our applications prefer block sizes of between 4 and 16 words on these machines (see table 4.2).

For comparison, we also include one prefetch cache organization, denoted by *Prefetch*(8,8). This cache uses a line size of 8 words and may fetch up to 8 lines (64 words), including the requested line, in one coherency transaction.

6.2 Network-Independent Analysis

In this section we compare the performance of the adjustable cache instances defined in section 6.1 against fixed line size caches, using network-independent metrics like the miss rate and *DTPR*. In our discussion we group applications according to these two metrics. For each application we describe the behavior of adjustable caches using the split and merge statistics, and the distribution of block transfers by block sizes.

6.2.1 *Minimal Miss Rate* Group

We call the first group of applications the *minimal miss rate* group with respect to the performance of adjustable caches. This group consists of *qsort*, *pgauss*, and *matmult*. For these applications the three instances of adjustable caches result in the three lowest miss rates among all the caches tested. The best *Fixed* miss rate is greater than the best adjustable cache miss rate by 42% for *matmult* (figure A.4.1), 15% for *pgauss* (figure A.6.1), and 13% for *qsort* (figure A.9.1). The amount of data transferred per reference was low for the adjustable cache with an initial block size of 16 words (*Vblock*(4,64,16)). For *qsort* this cache organization achieved *DTPR* equal to that of *Fixed*(8), whereas for the other two applications it resulted in *DTPR* substantially better (up to 20%) than that of *Fixed*(8). The other two adjustable caches achieved *DTPR* well below that of *Fixed*(32), even though their initial block size was 64 words. The cache with minimum block size of 8 words (*Vblock*(8,64,64)) even managed a lower *DTPR* than that of *Fixed*(16) in the case of *qsort* (figure A.9.2).

Splitting is the dominant form of block adjustment for these three applications. For each adjustable cache and each of these applications, the number of splits was higher than merges (see tables A.6, A.9, A.4). Moreover most of the transfers (between 60% and 80%) used the minimum block size on all three adjustable caches and two out of three applications - (*qsort* (figure A.9.4) and *matmult* (figure A.4.4)). In the case of *pgauss* (figure A.6.4), the dominance of the minimum block size in the transfer statistics is apparent for the adjustable cache with the smallest minimum block (4 words), which accounted for 77% of all transfers. The adjustable cache with minimum block size of 16 words performed close to 60% of all transfers with this block size for *pgauss*. The adjustable cache with minimum block size of 8 words used block sizes of 8, 16, and 32 words for a significant fraction of transfers (22%, 30% and 40% respectively).

For two applications - *qsort* (table A.9) and *pgauss* (table A.6) - the number of failed merges was high, which is the result of a high miss rate for these applications. That is, more misses give more opportunities to split or merge. With small split and merge thresholds, there are more merge attempts because blocks of the minimum size are used so often (and we cannot split a block of the minimal size!).

Although the minimum block size was used most frequently (except for *pgauss* and *Vblock*(8,64,64)) the larger blocks accounted for a significant fraction of transfers, at least 20%. This suggests that apart from fine-grain sharing in these three applications, there are data structures that prefer longer lines. That is, there is a certain degree of spatial locality that adjustable caches are able to exploit, as illustrated by the lower miss rates than those for fixed line size caches with short lines. On the other hand, fine-grain sharing makes longer cache lines impractical for these applications because of false sharing. Overall, adjustable caches are able to reduce false sharing by splitting down and using small blocks for some data and by exploiting spatial locality with larger blocks for other data. These two effects, i.e. elimination of false-sharing and exploitation of limited spatial locality, resulted in the lowest miss rates and reasonable *DTPR* among all caches tested.

The minimal miss rates achieved by adjustable caches for these applications ensure good performance (in relation to fixed line size caches) as captured by *MCPR*. This is especially true for high bandwidth machines, where the miss rate is the dominant factor.

6.2.2 *Reduced Dominant Metric* Group

We call the next group of applications the *reduced dominant metric* group with respect to the performance of adjustable caches. This group includes three applications - *plytrace*, *pmatmult*, and *sorbyc*. The name of this group suggests that the effect of adjustable caches for these applications depends on the parameters of the cache. That is, a cache with small initial and minimum blocks, like *Vblock*(4,64,16), reduces *DTPR* much more than the miss rate. Since this cache is intend for low bandwidth machines, this is the correct behavior. The other two adjustable caches, with large initial blocks, reduce the miss rate more than *DTPR*. Again, this is beneficial, as long as we want to use these caches in higher bandwidth machines.

For all three applications, an adjustable cache with the minimum block size of 4 words and initial block size of 16 words (*Vblock*(4,64,16)) results in a miss rate close to that of *Fixed*(16) and *DTPR* close to that of *Fixed*(8). The other two adjustable caches produce miss rates within 4% of the best fixed line size cache for *sorbyc* (figure A.10.1) and even better than the best fixed line size cache (by 5%) for *pmatmult* (figure A.8.1) and (by 8%) for *plytrace* (figure A.7.1). However, the *DTPR* was between that of *Fixed*(16) and *Fixed*(32) for both these caches on *plytrace* and *pmatmult*. Only in the case of *sorbyc* was *DTPR* close to that of *Fixed*(8), and only for the adjustable cache with a minimum block size of 8 words. The other adjustable cache, with minimum block size of 16 words, results in *DTPR* close to that of *Fixed*(16).

The split and merge statistics for *pmatmult* (table A.8) and *plytrace* (table A.7) show a surprisingly low number of adjustments, with few merges. The transfer statistics clearly show that two types of data exist in these applications - one which prefers larger blocks and the other which prefers the smallest possible blocks. For each adjustable cache, the smallest block accounted for between 30% and 60% of all transfers. On the other hand the largest block (64 words) was used for 30% (*pmatmult*) and 20% (*plytrace*) of the transfers for the two adjustable caches with initial block size equal to the maximum block size. The third cache, with an initial block size 16 words, was unable to merge blocks so all of these transfers were done with the initial block size. The exclusive use of small block sizes (between 4 and 16 words) explains why this particular cache results in higher miss rates for these two applications than the other two caches, which manage to exploit spatial locality by using larger blocks.

Sorbyc (figure A.10.4) is a bit different from the other two programs in this group because the smallest blocks dominate the transfer statistics (for two out of three adjustable caches), and there are few transfers with blocks of 32 or 64 words. Such a low utilization of large blocks is surprising in light of the fact that the number of merges is not much lower than the number of splits (table A.10). However, closer examination of this table shows that there were very few merge operations with blocks of 16 or 32 words. In the case of *sorbyc*, the tradeoff between the miss rate and *DTPR* is resolved mainly through the minimum block size. The cache with small minimum block size results in a higher miss rate, while the other two caches with larger minimum block sizes reduce the miss rate at the expense of *DTPR*.

6.2.3 *Reduced DTPR* Group

We call the next group of applications the *reduced DTPR* group with respect to the performance of adjustable caches. This group includes *gauss* and *mp3d* and consists of applications for which adjustable caches reduce *DTPR* at the expense of the miss rate.

For *mp3d*, the miss rates (figure A.5.1) of adjustable caches are comparable to that of fixed line size caches with line size equal to the minimum block size of the adjustable cache. *Vblock*(4,64,16) managed to reduce the miss rate of *Fixed*(4), but it is still higher than the miss rate of *Fixed*(8). Also the *DTPR* (figure A.5.2) of the adjustable caches is comparable to the fixed line size caches with line size equal to the minimum block size of the adjustable cache. In effect, adjustable caches degenerate to fixed line

size caches for *mp3d*. This is confirmed by the transfer statistics (figure A.5.4). Between 78% and 98% of all transfers were performed with the smallest block allowed.

For *gauss*, a substantial number of transfers (close to 20%) are performed with blocks larger than 16 words (figure A.2.4). Even the adjustable cache with an initial block size of 16 words managed to merge these initial size blocks and use blocks of 32 words in a substantial fraction of transfers. However, the minimum block size still has a big impact on the performance of adjustable caches, because it was used in between 76% and 84% of all transfers, depending on the cache organization.

For these applications even the small fraction of transfers that use large blocks help to improve the miss rate. The miss rate still depends strongly on the minimum block size (figure A.2.1), but is usually better than the miss rates of fixed line size caches with line size equal to the minimum block size of adjustable cache. For example, $Vblock(4,64,16)$ has a much lower miss rate than that of $Fixed(8)$, the miss rate of $Vblock(8,64,64)$ is equal to that of $Fixed(16)$, and $Vblock(16,64,64)$ has the miss rate of $Fixed(32)$. On the other hand, the data transferred per reference (figure A.2.2) for each adjustable cache closely follows the *DTPR* of the fixed line size cache with line size equal to the minimum block size.

Adjustable caches reduce *DTPR* for these two applications at the expense of the miss rate. This behavior is caused by too much splitting (especially in the case of *mp3d*), which in turn is a result of a low split threshold. We would expect adjustable caches to perform well on low bandwidth machines for this group of applications, but for very high bandwidth machines, some fixed line size caches may result in better performance (measured in *MCPR*) due to lower miss rates.

6.2.4 *Largest Block Preferred* Group

Finally, there is a group of applications which we call the *largest block preferred* group. This group includes three applications: *bsort*, *kmerge*, and *sorbyr*, which prefer large blocks regardless of the available bandwidth. For these applications, the goal of an adjustable cache should be to use the largest allowed block as often as possible (by merging blocks if the initial block size is different from the maximum block size) and avoiding splitting.

The two adjustable caches with a maximum block size of 64 words result in a miss rate equal to (for *sorbyr* and *kmerge*) or lower (for *bsort*) than the miss rate of the best fixed line size cache, which is always $Fixed(64)$ for these programs. The *DTPR* for these caches is close to that of $Fixed(64)$ for all three applications. There is also a limited amount of splitting for these two adjustable caches and practically no merging. A high fraction of all transfers (96% for *bsort*, 95% for *kmerge*, and 80% for *sorbyr*) are performed with a block size of 64 words.

The third adjustable cache, $Vblock(4,64,16)$, has a much harder task in the case of these applications, because its initial block size, 16 words, is much too small to result in good performance. Therefore, the only way for this cache to improve performance is via merging. For *sorbyr*, the program runs long enough (recall that this application trace was one of the longest) to allow this adjustable cache to merge blocks. Close to 50%

of all transfers are performed with blocks of 64 words (figure A.11.4), even though the initial block size is 16 words. However, this is much less than the other two adjustable caches, which use blocks of 64 words for more than 80% of all transfers. As a result *Vblock*(4,64,16) produces a miss rate close to that of *Fixed*(32) (figure A.11.1), instead of *Fixed*(64), but manages to cut *DTPR* down to that of *Fixed*(4).

For the other two applications in this group, ***bsort*** and ***kmerge***, merging is not very successful in *Vblock*(4,64,16). Both of these applications have short traces and there are not enough opportunities to merge. In effect, close to 70% of all transfers are produced with the initial block size of 16 words (figures A.1.4 and A.3.4). Longer blocks (especially blocks of 32 words) are responsible for the remaining 30%, which indicates that there is ongoing merging activity in both of these cases. As a result, the miss rate of *Vblock*(4,64,16) is 24% lower than that of *Fixed*(16) for ***bsort*** (figure A.1.1) and 16% lower for ***kmerge*** (figure A.3.1). However, both miss rates are substantially higher than that of *Fixed*(32). The *DTPR* remains flat for all caches since these programs have excellent spatial locality.

6.3 Effect on *MCPR*

In this section we discuss the performance of adjustable caches expressed as the mean cost per reference (*MCPR*) for one particular machine. Of the five machines described in chapter 4, we select the machine with bandwidth equal to 40 MB/sec. This machine is denoted in our notation by $M(50,10)$, i.e. $F_{latency}$ is equal to 50 and $F_{bandwidth}$ is equal to 10. We believe this choice represents a reasonable machine of today. The performance of adjustable caches for the remaining machines is investigated in chapter 7.

Table 6.1 gives the mean cost per reference relative to the adjustable cache *Vblock*(4,64,16) for five fixed line size caches (with line sizes between 4 and 64 words) without prefetching, and two prefetching caches, *Prefetch*(8,8) and *Prefetch*(16,4). If the performance of a fixed line size cache for a given application is described by a number greater than 1, this indicates that *Vblock*(4,64,16) resulted in better performance than that fixed line size cache. On the other hand, if this number is less than 1, the fixed line size cache was better.

Our first observation is that nearly all numbers in this table are greater than 1, i.e. the adjustable cache is better in most of the cases. The two significant exceptions are ***bsort*** and ***kmerge***, for which caches with long lines perform better than *Vblock*(4,64,16) by up to 13%. This is an effect of short traces, however, as noted in section 6.2.4. That is, both of these programs are too short to allow merging to minimize the effect of the initial block size, which is 16 words. As a result, this adjustable cache has *MCPR* only slightly better than *Fixed*(16) (6% for ***bsort*** and 4% for ***kmerge***). However, for long traces like ***sorbyr***, merging is effective, and *Vblock*(4,64,16) results in a mean cost per reference within 1% of *Fixed*(64).

A second observation is that all fixed line size caches without prefetching suffer at least a 29% increase in *MCPR* for at least one application. Caches with short lines like

Table 6.1: $M(50,10)$ Machine - *MCPR* Relative to $Vblock(4,64,16,(1,1))$

Application	Cache						
	Fxd(4)	Fxd(8)	Fxd(16)	Fxd(32)	Fxd(64)	Prftch(8,8)	Prftch(16,4)
matmult	1.15	1.09	1.18	1.67	2.89	1.12	1.24
pgauss	1.18	1.18	1.37	2.15	3.84	1.16	1.37
qsort	1.17	1.10	1.25	1.77	3.56	1.08	1.29
plytrace	1.11	1.02	1.08	1.38	2.31	1.06	1.15
pmatmult	1.30	1.13	1.07	1.14	1.42	1.08	1.06
sorbyc	1.03	1.01	1.07	1.22	1.55	1.00	1.07
gauss	1.09	1.05	1.02	1.07	1.19	1.03	1.02
mp3d	1.06	0.99	1.16	1.63	3.01	1.13	1.29
bsort	1.76	1.29	1.06	0.95	0.89	1.19	1.02
kmerge	1.79	1.29	1.04	0.92	0.87	1.21	1.01
sorbyr	1.33	1.14	1.05	1.01	0.99	1.13	1.05

Fixed(4) and *Fixed*(8), suffer on the *largest block preferred* group, especially *bsort* and *kmerge*. For *Fixed*(4), both of these applications exhibited a 70% increase in *MCPR*; for *Fixed*(8), the increase was 29%.

Caches with longer lines suffer on the *minimal miss rate* group of applications, for which the adjustable cache managed to eliminate false sharing. For *Fixed*(16), the increase in *MCPR* was 37%, 25% and 18% for the three applications belonging to this group. These percentage increases are much larger for *Fixed*(32), (115%, 77% and 67% respectively). And for *Fixed*(64), the mean cost per reference increases by a factor of 3 or so for each of these applications when compared against $Vblock(4,64,16)$.

Prefetching caches offer little improvement. $Prefetch(16,4)$ results in a higher *MCPR* than *Fixed*(16) for the *minimal miss rate* group of applications. Although $Prefetch(8,8)$ is an improvement over *Fixed*(8), the 29% increase in *MCPR* introduced by *Fixed*(8) on *kmerge* and *bsort* remains significant (21% for *kmerge* and 19% for *bsort*). The performance of both prefetching caches, as well as the miss rates and *DTPR* data for $Prefetch(8,8)$ (see appendix A), show clearly that prefetching has limited effect on all metrics. That is, prefetching caches behave like fixed line size caches without prefetching. Therefore, prefetching with shorter lines, for example $Prefetch(1,64)$, would result in poor performance, i.e. close to that of *Fixed*(1). The reason for this anomaly is that a prefetching cache does not have enough flexibility to correctly predict the sharing patterns. That is, it prefetches blocks only when there is a write to a *modified* block or a read of a *read-shared* block. However, for *kmerge* and *bsort*, more than 90% of all coherency transactions are of another type (i.e. write to *read-shared* or read of a *modified* block). In effect, in more than 90% of the misses, the prefetching cache operates like a simple fixed line size cache for both of these applications. One remedy would be to prefetch during this 90% of coherency transactions. This approach would cause the prefetching cache to behave like a fixed line size cache with long lines

however, which is unacceptable for programs with fine-grain sharing.

The two caches most competitive with $Vblock(4,64,16)$ are $Fixed(8)$ and $Prefetch(8,8)$. For $Fixed(8)$, we have 2 programs that suffer a 29% increase in *MCPR*, and 6 out of 11 programs suffer at least a 10% increase compared to $Vblock(4,64,16)$. On the other hand, $Fixed(8)$ is better only in the case of *mp3d*, and only by 1%. For $Prefetch(8,8)$, two programs suffer a 20% or so increase in *MCPR*, and 6 out of 11 suffer at least a 12% increase. Also, there is no program for which $Prefetch(8,8)$ performs better than the adjustable cache. We note also that the biggest increases in *MCPR* were recorded for *kmerge* and *bsort* for both of these caches, even though the adjustable caches were not able to exploit fully the advantages of merging for these programs because they were too short. We conclude that for programs with a strong preference for large blocks, and sufficient duration to allow a high degree of merging to occur, both $Fixed(8)$ and $Prefetch(8,8)$ would result in a much higher *MCPR* than the adjustable cache. This conclusion makes both $Fixed(8)$ and $Prefetch(8,8)$ even less competitive with $Vblock(4,64,16)$ then is suggested by table 6.1.

6.4 Cache Size

This section discusses the size of the cache needed to execute our applications. We still assume fully associative caches, which eliminates conflict misses. We want to know the minimum cache space needed to store all referenced data (plus any additional cache control bits) while avoiding capacity misses and reusing cache lines invalidated on coherency transactions. Our metric is given in bits of cache space needed per reference to allow comparison across applications. This metric also provides a rough estimate of how much space overhead is introduced by adjustable caches.

For fixed line size caches (with or without prefetching), the value of the metric is the maximum number of cache lines used in the execution of a given program times the length of a cache line. This length includes tag bits and additional control bits, like the *modify* bit and state bits.

For adjustable caches we compute the size for two possible implementations. The first size of interest is the maximum size of an adjustable cache implemented as a fully-associative cache with line size $MaxBlockSize$ (i.e. the *max-block* implementation described in section 5.2.1). The total line size in bits includes tag bits (for $MinBlockSize$), size bits, state bits, the *modify* bit, two LU bits, and split-merge counter bits (4).

The second size of interest is the size of an adjustable cache implemented as a fully-associative cache, but with blocks spanning multiple lines, where the size of each line is equal to $MinBlockSize$ (i.e. the *min-block* implementation described in section 5.2.1). The control part of each line in this implementation is one bit wider than for the *max-block* implementation (called the *head-tail* bit), because we now have to distinguish between head and tail lines. The size of the *min-block* implementation is the maximum number of lines (where the data part size is equal to $MinBlockSize$) occupied during program execution times the total line length. The total line length includes both data and control parts for each line, so we account for additional control fields in lines used to hold blocks larger than $MinBlockSize$.

These two implementations represent two very different choices. The *max-block* implementation is simpler but may result in low cache utilization if the average length of a data block is much smaller than $MaxBlockSize$. On the other hand, this implementation results in good cache utilization if most of the blocks are large and only a few are small. The *min-block* implementation is more challenging for the hardware designer because blocks spanning multiple lines introduce an additional level of complexity. However, the *min-block* implementation may be much more space efficient than the *max-block* implementation in some situations.

The conditions under which a given implementation is more space efficient become apparent if we compute how much space is needed on the average to house a unit of data (say one word). For the *min-block* implementation we use $ControlSize + MinBlockSize$ space to keep $MinBlockSize$ data, where $ControlSize$ is the size of the control part of a cache line, including the tag. Therefore the space overhead per word of data in the *min-block* implementation is given by:

$$\frac{ControlSize + MinBlockSize}{MinBlockSize} = 1 + \frac{ControlSize}{MinBlockSize}$$

For the *max-block* implementation we use $ControlSize + MaxBlockSize$ space to keep each block of data. We assume that the sizes of control parts are very similar for both implementations. If $AvgBlockSize$ is the average block size then the space overhead per one word of data is given by:

$$\frac{ControlSize + MaxBlockSize}{AvgBlockSize} = \frac{MaxBlockSize}{AvgBlockSize} + \frac{ControlSize}{AvgBlockSize}$$

We see that this overhead depends on the maximum block size in the *max-block* implementation, but is independent of the maximum block size in the *min-block* implementation. If the average block size is close to the minimum block size, then $\frac{MaxBlockSize}{AvgBlockSize}$ is much larger than 1 and in effect we need much more space for the *max-block* implementation. On the other hand, if the average block size is close to the maximum block size, then $\frac{MaxBlockSize}{AvgBlockSize}$ is close to 1 and $\frac{ControlSize}{AvgBlockSize}$ is smaller than $\frac{ControlSize}{MinBlockSize}$, so the *max-block* implementation may be more efficient.

The good news is that in many cases the space overhead of adjustable caches is reasonable regardless of the implementation. These cases include two instances of adjustable caches in which the initial block size is equal to the maximum block size: $Vblock(8,64,64)$ and $Vblock(16,64,64)$. These two caches need relatively small cache space for our application suite, sometimes comparable or even smaller than some of the *Fixed* caches, and always within 40% of the best *Fixed* caches for the following applications: *matmult*, *pmatmult*, *sorbyc*, *gauss*, *bsort*, *kmerge*, *sorbyr*. These are the programs that keep most of the data in large blocks (although it does not necessarily mean that most of the transfers are done with such blocks). And large blocks make the *max-block* implementation space efficient. For this implementation the largest increase in the cache space needed by either of these two adjustable caches over the best *Fixed* cache is 39% (in the case of *pmatmult*, figure A.8.3). Also, the minimum block sizes for these two caches are large enough (8 and 16 words, i.e. 32 and 64 bytes) to make the space overhead resulting from additional control parts of $MinBlockSize$ cache lines

manageable for the *min-block* implementation. For this implementation, the largest increase in the cache space needed by either of these two adjustable caches over the best *Fixed* cache is 37% (again in the case of *pmatmult*).

For other applications (which use small blocks a lot), the space overhead of these two adjustable caches (compared against *Fixed* caches) depends very much on the implementation. For the *min-block* implementation, the cache space increase is less than the 37% recorded for the previous group of applications because of the increased use of small blocks. For the *max-block* implementation some programs resulted in a large amount of wasted data space. Compared to the most space efficient *Fixed* cache, these two adjustable caches need 150% more space for *pgauss* (figure A.6.3), 168% for *mp3d* (figure A.5.3), 170% for *plytrace* (figure A.7.3) and up to 358% for *qsort* (figure A.9.3. All these overheads result from long cache lines (64 words) and a much smaller (close to the minimum block size) average block size produced by heavy splitting.

The third adjustable cache we considered, $Vblock(4, 64, 16)$, introduces even more overhead in the *max-block* implementation, and this is true for practically all applications. As in the previous examples, heavy use of small blocks caused by splitting wastes a lot of space, but there is another factor which makes this implementation especially wasteful: The initial block size is different from the maximum block size, and we waste data space by default. The minimum overhead this cache introduces over the best *Fixed* cache is equal to 138% (in the case of *gauss*, figure A.2.3), while the maximum overhead is equal to 670% (in the case of *qsort*, figure A.9.3). On the other hand, the *min-block* implementation of $Vblock(4, 64, 16)$ results in moderate overhead. The largest overhead over the most space-efficient *Fixed* cache, equal to 46%, was recorded for *qsort* and the adjustable cache with a minimum block size of 4 words. This maximum overhead is 37% when the minimum block size is 8 words, and 28% when it is 16 words. We note that this maximum overhead is reduced with larger minimum block sizes.

From this analysis we conclude the following: the *min-block* implementation of adjustable caches introduces manageable space overhead (up to 46%) compared with the best *Fixed* caches. The *max-block* implementation offers reasonable cache utilization only if the initial block size is equal to the minimum block size, and only if small blocks are rarely used.

6.5 Summary

In this chapter we proposed three instances of adjustable caches intended to cover wide range of multiprocessors with various levels of network bandwidth. We compared these three adjustable caches with fixed line size caches by running the simulations for all our applications. For three out of eleven applications all three adjustable caches managed to reduce the miss rate below that of the best fixed line size cache. The best fixed line size cache had a miss rate higher by between 13% and 43% compared to the best adjustable cache. At the same time the amount of data transferred by the adjustable caches was also low, usually close to the *DTPR* recorded by the fixed line size caches with line size equal to the minimum block size.

For three other applications, two adjustable caches reduced the miss rate at the expense of *DTPR*, while the other adjustable cache, (with a small minimum block size) reduced *DTPR* at the expense of the miss rate.

There were two applications for which all adjustable caches split down to the minimum block size quickly. In effect, the adjustable caches performed like fixed line size caches with the line size equal to the minimum block size. Finally, for the three applications preferring large blocks, the adjustable caches correctly avoided splitting and tried to merge up to the maximum block size. The success of merging depended on how long the program ran.

Overall, the adjustable caches behaved in many different ways depending on the initial parameters and the application. In a few cases very little adjustment took place, because the initial block size was the optimal one. In a few another cases, merging was the dominant method of adjustment. For most of the applications both splitting and merging took place, however the number of splits was usually much higher than the number of merges, which is in part a consequence of large initial blocks.

We continued the comparison study by computing the mean cost per reference for all our applications on 40 MB/sec machine with varying cache organizations. In this part of the study we compared an instance of the adjustable cache that was well-suited to this machine against all fixed line size caches and two prefetching caches. It turned out that the adjustable cache was a winner in nearly all cases, i.e. it resulted in the lowest mean cost per reference. The most competitive fixed line size cache was better for only one application (and only by 1%), whereas for six out of eleven applications, it was worse by more than 10%, and for two applications by 29%. The most competitive prefetching cache performed worse for all applications, by at least 12% for 6 applications, and up to 21% in the worst case. Prefetching did not help much because, in most of the cases, there was not enough of it.

For all applications the number of splits and merges performed by any adjustable cache was low compared to the number of references or even to the number of misses. Therefore, even more expensive split and merge operations would not change the mean cost per reference substantially. However, the number of failed merges was high for some programs, and therefore it is important to keep the cost of a failed merge low.

The *min-block* implementation of adjustable caches introduces limited space overhead over the most space efficient fixed line size cache. For all three adjustable caches and all applications this overhead was no higher than 46%. The *max-block* implementation is space efficient only if the initial block size is equal to the maximum block size, and only if small blocks are rarely used.

7 Selecting an Instance of the Adjustable Block Size Cache Organization

This chapter discusses how to select an adjustable cache for a particular machine and how much improvement over fixed line size caches we can expect to achieve.

In section 7.1 we examine the effect of the parameters used to control changes in block size on the performance of adjustable block size caches.

In sections 7.2, 7.3, and 7.4 we discuss the selection of an adjustable cache for $M(50,*)$ machines. That is, we vary the network bandwidth, while keeping latency constant at 50 cycles, and show how to select an adjustable cache for each such machine. We also evaluate how much adjustable caches can improve the mean cost per reference relative to fixed line size caches depending on the available bandwidth.

Finally, in section 7.5 we consider how adjustable caches would perform on machines with latency higher than 50 cycles.

7.1 Effects of Adjustable Block Size Cache Parameters

In this section we discuss the tradeoffs to be considered when selecting the split and merge threshold values, the initial block size, the maximum and minimum block sizes, and the amount of reference history maintained in the split-merge counter. We also quantify the effect of each of these parameters in our simulations. All *MCPR* results in this section are given for machines with latency of 50 cycles.

7.1.1 Split and Merge Thresholds

The split and merge thresholds determine how quickly the block size adjusts to reference behavior. By choosing small values for these thresholds, we can encourage both splits and merges to occur. In our simulations, we set the split threshold to 1 and the merge threshold to 1. We chose small values for both to encourage changes in the block size.

Additional simulations showed that small changes in the split threshold do not alter the results significantly. In these experiments we used $Vblock(8,64,64,(1,1))$ as the base cache, because this cache should be sensitive to changes in the split threshold.

Setting the split threshold to 2 instead of 1 resulted in a 6% increase in *DTPR* and the same miss rate for *matmult*. When the split threshold was 3 instead of 1 the *DTPR* increased 9%, while the miss rate still remained unchanged. For other applications this pattern was similar, i.e. an increase in the split threshold caused small increases (up to 9%) in *DTPR* and had very little effect on the miss rate (usually it was reduced by no more than 1%). One exception was *sorbyr*, for which a split threshold of 2 reduced the miss rate by 9%. Apparently a split threshold of 1 makes this application too sensitive to splitting (recall that *sorbyr* prefers large blocks).

We used $Vblock(4, 64, 16, (1, 1))$ as the base cache to test the impact of the merge threshold. Small changes in the merge threshold affect performance only for programs that prefer large cache blocks. Raising the merge threshold to 2 increased the miss rate by 20% for *bsort*, 10% for *kmerge* and only 2% for *sorbyr*. The data transferred per reference remained unchanged. For other applications, raising the merge threshold had very little effect.

In general the split threshold has lower impact on the performance of adjustable caches than the merge threshold. Since it is much more difficult to merge than to split, the merge threshold should be low to facilitate merging.

7.1.2 Initial Block Size

The initial block size is an important parameter for the adjustable block size cache because it sets the starting point from which we attempt to converge to the optimal block size. The smaller the number of steps (in splits or merges) between the initial block size and the optimal block size for a given piece of data within an application, the better the performance of the adjustable block size cache.

It follows from table 4.2 that the optimal cache line size depends on two factors: application characteristics and the network specification. Selection of the initial block size for a given machine depends on the set of applications we expect to run on it. The lower the bandwidth, the wider the range of optimal block sizes, and the more difficult the selection of the initial block size.

For low bandwidth machines we selected an initial block size equal to 16 words, even though this selection is far from optimal for applications that prefer large blocks. The majority of applications prefer blocks smaller than 16 words, and only two out of eleven of our applications prefer cache lines of 16 words (table 4.2) for these machines. Clearly the choice of initial block size is a compromise between applications with good spatial locality and those with fine-grain sharing.

Since merging is a much slower process than splitting, we generally recommend large initial blocks. In the *max-block* implementation of adjustable caches, the initial block size should be equal to the maximum block size, because cache utilization is very poor otherwise (see section 6.4). Additionally, large initial blocks make programs with good spatial locality perform better and we should encourage the use of such programs.

At times we need to start with larger blocks to exploit a preference for two or more block sizes by one application. For example, an initial block size of 16 words proved to

be better in terms of *MCPR* than an 8-word initial block size for *matmult* on the lowest bandwidth machine, even though *Fixed*(8) performed better than *Fixed*(16). We can see from figure A.4.4 that, when the initial block size is 16 words, the vast majority of all transfers are of size 4 words or 16 words. Although only 3% of all transfers use blocks of 32 words or larger, these transfers account for 12% of all the data transferred. If we change the initial block size to 8 words, too few merge operations occur. As a result, only a few transfers use a cache block of 16 words or larger. It is this lack of merge operations that causes a cache with an initial block size of 8 words to perform worse than a cache with an initial block size of 16 words, even though *Fixed*(8) performs better than *Fixed*(16).

7.1.3 Maximum Block Size

For the *max-block* implementation of adjustable caches, the maximum block size is determined by the size of a cache line. A single cache line should be big enough to exploit spatial locality, but not so big that most of the cache line goes unused. Thus, there is a tradeoff between the performance benefits of a larger cache line and the cost of unused cache space.

For programs that prefer large blocks the maximum block size defines the limit of how well a given adjustable cache can perform. Additionally, if the initial block size is equal to the maximum block size, the performance of an adjustable cache for these programs is very close to that of the fixed line size cache with line size equal to the maximum block size. Therefore, to see the effect of a smaller maximum block size on an adjustable cache with the initial block size equal to the maximum block size, it is sufficient to compare the performance of *Fixed* caches. For example, reducing the maximum block size from 64 words to 32 words increases *MCPR* between 5% and 8% for *qsort*, 4% and 10% for *kmerge* and up to 3% for *sorbyr*. Reducing the maximum block size from 32 words to 16 words would increase *MCPR* even more due to the reverse effect of diminishing improvements, as discussed in chapter 4.

When the initial block size is smaller than the maximum block size, then reducing the maximum block size has one more effect: Apart from limiting the performance of programs preferring large blocks, it may now avoid some merges and thereby reduce *DTPR* at the cost of the miss rate for programs preferring small blocks, especially if the merge threshold is low. For example, when the maximum block size is set to 16 words instead of 64 words in *Vblock*$(4, 64, 16, (1, 1))$, the miss rate increases by 10% for *qsort*, but the *DTPR* is reduced equally.

This result also points out the relationship between the merge threshold and the maximum block size. When the merge threshold is low (as it was in our simulations), additional merges take place, as in the case of *qsort*. For these programs, a large maximum block size simply increases the opportunity for merge operations. Shrinking the maximum block size or raising the merge threshold would prevent merges, but either alternative would adversely affect the performance of other programs that require merge operations to produce an appropriate block size. Raising the merge threshold makes it difficult for programs such as *kmerge*, *bsort*, and *sorbyr* to quickly adjust the block size

upwards, while shrinking the maximum block size prevents these programs from ever achieving the desired larger block sizes.

7.1.4 Minimum Block Size

The choice of minimum block size depends on the following factors:

- How much fine-grain sharing do we expect in our applications ?
- What is the ratio of latency to bandwidth? The lower the bandwidth, the smaller the minimum block size needed (see table 4.2).
- If the adjustable cache is implemented with the *min-block* approach, how much space overhead are we willing to bear? The smaller the minimum block size, the more overhead incurred by the *min-block* implementation. For the *max-block* implementation, the minimum block size together with the maximum block size defines the upper bound on wasted cache space.

To see the effect of the block size range on performance, we increased the minimum block size of $Vblock(4,64,16,(1,1))$ to 8 and reran our simulations for a number of traces. *Mp3d* is an example of a program which may benefit from the higher minimum block size, depending on the host machine. The miss rate was reduced by 17%, but *DTPR* shot up by 42% with the increased minimum block size. Recall (figure 4.8) that $Fixed(8)$ is the best fixed line size cache for 50/5 and 50/10 machines. Clearly there is too much splitting (or the minimum block size is too small) when using $Vblock(4,64,16,(1,1))$ for these machines. On the other hand, on the 50/20 machine (table 7.4) this adjustable cache performs quite well, a little better than the optimal fixed line size cache, $Fixed(4)$. In the case of *gauss* the effect of the increased minimum block size was similar; the miss rate decreased by 6% and *DTPR* increased by 37%. For other programs, like *qsort*, *matmult* and *pmatmult*, the miss rate did not change, but *DTPR* increased substantially, up to 40%.

The relationship between the minimum block size and split threshold is analogous to the relationship between the maximum block size and the merge threshold. When the split threshold is low and the minimum block size is small, too many splits can occur (as in *mp3d* and *gauss*). This situation arises for example when a program initializes adjacent data from different processors (which causes splits), but then exhibits good processor locality for subsequent accesses. On the other hand, some programs like *qsort* perform best with a small block size, and a low split threshold that allows blocks to shrink quickly. An increase in the split threshold or the minimum block size penalizes these programs.

7.1.5 Reference History

The split-merge counter records the reference history to a cache block as it moves from one cache to another. We would like the information in the counter to be as current

as possible without resorting to time-stamps or aging. We would also like to ensure that slight changes in reference behavior do not cause an excessive number of splits or merges to occur.

One way to control the ease with which the cache adjusts to changes in reference behavior is to limit the size of the split-merge counter. With a small counter, reference information is quickly replaced, ensuring that information accumulated during an earlier phase of execution cannot prevent splits or merges during a subsequent phase. With a larger counter, the cache is less sensitive to short-term reference patterns. Of course the counter must be big enough to hold the integers between the merge threshold and the split threshold, but a larger counter can also record reference information that temporarily exceeds the thresholds, such as occurs when reference information from several copies is brought together.

All our previous experiments used a 4-bit split-merge counter. To test the sensitivity of our results to this parameter we reran our experiments with a 3-bit counter and a 5-bit counter for $Vblock(4,64,16,(1,1))$. All other parameters remained the same. Several applications including *bsort*, *matmult*, and *kmeans*, showed no sensitivity to the size of the split-merge counter, which suggests there are no changes in reference behavior during execution of these applications. For *mp3d*, the miss rate increased by 1% and *DTPR* decreased by 1% when a 3-bit counter was used. With the 5-bit counter the opposite effect was observed. The miss rate decreased by 1% and *DTPR* increased by 2%. Similar changes were observed for *qsort*.

We conclude from these experiments that the size of the split-merge counter does not significantly influence the performance of adjustable caches. Only programs with fine-grain sharing are sensitive to changes in the size of this counter. Reduction of the counter size increased the miss rate and decreased *DTPR*, whereas a large counter had the opposite effect. However, the observed changes were minimal, on the order of 1% change for both the miss rate and *DTPR*. As a result the use of small counters (say, 3 or 4 bits) not only save space, but also provide good performance across all applications.

7.2 High Bandwidth Machines

In this section we compare one instance of the adjustable cache organization with fixed line size caches and prefetching caches using *MCPR* for two machines: one with infinite bandwidth ($M(50,0)$ machine), and one with high finite bandwidth ($M(50,1)$ machine).

For high bandwidth machines the range of optimal cache line sizes is quite narrow. Recall table 4.2, where this range included only three line sizes: 16, 32 and 64 words for the two machines with the highest bandwidth.

On the infinite bandwidth machine, six out of eleven applications preferred a line size of 64 words. The adjustable cache we recommend for this machine is $Vblock(16,64,64,(1,1))$. An initial block size of 64 words allows for optimal performance for these six applications. Split threshold equal to 1 makes the adjustable cache very responsive to the recent reference pattern and allows applications that prefer blocks of 16 words to perform well.

On the infinite bandwidth machine, the miss rate is the dominant metric influencing the mean cost per reference. The amount of data transferred (*DTPR*) does not matter in this case because we have infinite bandwidth. The incurred cost per reference depends directly on the miss rate and network latency. Infinite bandwidth machines are the ultimate test case for adjustable caches, because any improvement in performance comes from cuts in the miss rate only, not *DTPR*. And to reduce the miss rate, we need to reduce false sharing.

Table 7.1: $M(50,0)$ Machine - *MCPR* Relative to $Vblock(16,64,64,(1,1))$

Application	Cache						
	Fxd(4)	Fxd(8)	Fxd(16)	Fxd(32)	Fxd(64)	Prftch(8,8)	Prftch(16,4)
matmult	1.21	1.09	1.04	1.13	1.23	1.06	1.03
pgauss	1.49	1.27	1.13	1.16	1.20	1.17	1.10
qsort	1.60	1.29	1.14	1.09	1.25	1.18	1.11
plytrace	1.41	1.17	1.06	1.03	1.11	1.11	1.03
pmatmult	1.58	1.28	1.12	1.04	1.01	1.21	1.09
sorbyc	1.11	1.04	1.01	1.00	1.00	1.03	1.01
gauss	1.15	1.08	1.02	1.00	0.99	1.06	1.01
mp3d	1.47	1.16	1.01	0.94	1.01	1.15	1.00
bsort	2.27	1.59	1.25	1.08	1.00	1.44	1.19
kmerge	2.56	1.73	1.31	1.10	1.00	1.59	1.25
sorbyr	1.38	1.18	1.07	1.02	1.00	1.16	1.07

Table 7.1 gives the mean cost per reference of all applications on the infinite bandwidth machine. The *MCPR*, given for five *Fixed* and two prefetching caches, is relative to $Vblock(16,64,64,(1,1))$, in a format similar to table 6.1.

Fixed line size caches with shorter lines (4 or 8 words) are not competitive in the presence of infinite bandwidth because they result in high miss rates. Compared to $Vblock(16,64,64,(1,1))$ they introduce substantial overhead. This overhead is especially noticeable for programs with good spatial locality, like *bsort* and *kmerge*, for which the *MCPR* of $Fixed(4)$ is at least 120% higher than that of $Vblock(16,64,64,(1,1))$. For $Fixed(8)$, the *MCPR* is at least 59% higher than $Vblock(16,64,64,(1,1))$ for these two programs. $Prefetch(8,8)$ does not improve *MCPR* significantly; it was still higher by 59% (*kmerge*) and 44% (*bsort*).

$Fixed(16)$ operates with relatively short cache lines of 16 words and introduces overhead in *MCPR* equal to 31% for *kmerge* and 26% for *bsort*. $Prefetch(16,4)$ reduces this overhead to 25% for *kmerge* and 19% for *bsort*.

Fixed line size caches with longer line sizes of 32 and 64 words result in *MCPR* closer to that of $Vblock(16,64,64,(1,1))$ for both *kmerge* and *bsort*. However, these fixed line size caches introduce more overhead for the *minimal miss rate* group of applications, which consists of *qsort*, *pgauss* and *matmult*. This confirms our observation that the

miss rate is the most important metric for high bandwidth machines. Compared to *Vblock*(16,64,64,(1,1)), *Fixed*(64) introduces at least 20% overhead for these three applications. *Fixed*(32) is the most competitive fixed line size cache; the highest overhead over *Vblock*(16,64,64,(1,1)) introduced by *Fixed*(32) is 16% (for *pgauss*). The overhead is 13% for matmult and 9% for *qsort*. For the *minimal miss rate* group of applications, *Vblock*(16,64,64,(1,1)) manages to reduce the mean cost per reference only by eliminating the false sharing introduced by a fixed line size of 32 words. The prefetching effect of a large initial block size (equal to 64 words) is not very useful here because all three applications prefer blocks smaller than 64 words. This prefetching effect is crucial for both *kmerge* and *bsort* however.

Overall for five out of eleven applications the most competitive fixed line size cache, *Fixed*(32), introduces overhead in *MCPR* between 8% and 16%. For four other applications this overhead is between 0% and 4%, and for one application (*mp3d*) *Fixed*(32) reduced *MCPR* by 6%. This failure of the adjustable cache to out-perform a fixed line size cache is due to the low split threshold and the small minimum block size, both of which reduce the *DTPR* at the expense of the miss rate.

Table 7.2: $M(50,1)$ Machine - *MCPR* Relative to *Vblock*(16,64,64,(1,1))

Application	Cache						
	Fxd(4)	Fxd(8)	Fxd(16)	Fxd(32)	Fxd(64)	Prftch(8,8)	Prftch(16,4)
matmult	1.20	1.07	1.04	1.15	1.34	1.06	1.04
pgauss	1.33	1.15	1.06	1.18	1.44	1.07	1.04
qsort	1.47	1.21	1.10	1.14	1.52	1.11	1.08
plytrace	1.33	1.12	1.03	1.04	1.21	1.07	1.01
pmatmult	1.53	1.25	1.10	1.04	1.06	1.19	1.08
sorbyc	1.09	1.03	1.01	1.02	1.05	1.02	1.01
gauss	1.15	1.08	1.02	1.00	1.00	1.06	1.01
mp3d	1.38	1.11	1.00	1.01	1.25	1.11	1.01
bsort	2.24	1.58	1.25	1.08	1.00	1.42	1.18
kmerge	2.50	1.70	1.30	1.10	1.00	1.56	1.24
sorbyr	1.38	1.17	1.07	1.02	1.00	1.16	1.07

If the bandwidth is high but finite, as in the $M(50,1)$ machine (which corresponds to 400 MB/sec given 100 MHz processors), the relative performance of *Vblock*(16,64,64,(1,1) is even better (table 7.2). The best fixed line size cache is still *Fixed*(32), but the overhead in *MCPR* over that of *Vblock*(16,64,64,(1,1) is now larger than for the unlimited bandwidth machine. For *minimal miss rate* applications this overhead is now 18% instead of 16% for *pgauss*, 15% instead of 13% for *matmult* and 14% instead of 9% for *qsort*. The adjustable cache was able to improve its performance due to a reduction in *DTPR*. However, there is no change in overhead for applications with good spatial locality because the amount of data transferred was the same for *Fixed*(32) and

Vblock(16,64,64,(1,1) (see figures A.1.2, A.3.2, A.11.2).

One more important observation is that even if the available bandwidth is high, the penalty introduced by 64 word lines is substantial for programs with false sharing. If the adjustable cache manages to reduce false sharing (as in the *minimal miss rate* applications), *Fixed*(64) introduces 20-25% overhead with infinite bandwidth and 34-52% with 400 MB/sec bandwidth. We conclude that even for high bandwidth machines, we cannot accept long cache lines if we also want to run programs with fine-grain sharing efficiently. And infinite bandwidth does not eliminate this problem.

7.3 Medium Bandwidth Machines

In this section we discuss the mean cost per reference for a medium bandwidth machine ($M(50,5)$), which corresponds to 80 MB/sec. For this machine, five out of eleven applications prefer a cache line size of 8 words (table 4.2), three prefer a cache line of 16 words, and the remaining three prefer a line size of 64 words.

We use *Vblock*(8,64,64,(1,1)) for this machine. This selection makes explicit our preference for applications with good spatial locality, because large initial blocks of 64 words allow for optimal performance of *Vblock*(8,64,64,(1,1)) with these applications. The low split threshold should allow applications with fine-grain sharing to split down to a block size of 8 words. The relatively high bandwidth of the host machine should allow this adjustable cache to perform close to the optimal fixed line size cache, even for applications with fine-grain sharing.

Table 7.3: $M(50,5)$ Machine - *MCPR* Relative to *Vblock*(8,64,64,(1,1))

Application	Cache						
	Fxd(4)	Fxd(8)	Fxd(16)	Fxd(32)	Fxd(64)	Prftch(8,8)	Prftch(16,4)
matmult	1.18	1.08	1.10	1.40	2.10	1.08	1.13
pgauss	0.94	0.88	0.91	1.26	2.02	0.84	0.91
qsort	1.20	1.04	1.07	1.35	2.39	1.00	1.08
plytrace	1.09	0.95	0.94	1.09	1.61	0.95	0.97
pmatmult	1.41	1.19	1.09	1.10	1.25	1.13	1.07
sorbyc	1.06	1.02	1.04	1.12	1.29	1.01	1.04
gauss	1.10	1.05	1.00	1.01	1.07	1.02	0.99
mp3d	1.19	1.02	1.06	1.32	2.19	1.10	1.14
bsort	2.11	1.51	1.22	1.07	1.00	1.38	1.16
kmerge	2.28	1.59	1.25	1.09	1.01	1.48	1.20
sorbyr	1.36	1.17	1.07	1.02	1.00	1.15	1.06

Table 7.3 gives the mean cost per reference of all applications on the $M(50,5)$ machine. The *MCPR*, given for five *Fixed* and two prefetching caches, is relative to *Vblock*(8,64,64,(1,1)).

As expected the adjustable cache is equal to $Fixed(64)$ for programs with good spatial locality like *bsort*, *kmerge* and *sorbyr*. If we consider good performance on these programs essential for a cache associated with the $M(50,5)$ machine, then we can immediately eliminate $Fixed(4)$ (up to 128% overhead over $Vblock(8,64,64,(1,1))$, $Fixed(8)$ (up to 59% overhead) and $Prefetch(8,8)$ (up to 48% overhead).

$Fixed(64)$ cannot be taken seriously for this machine if we want to run applications with fine-grain sharing. This cache resultes in at least 100% overhead over $Vblock(8,64,64,(1,1))$ for four applications. We can also eliminate $Fixed(32)$, but now the overheads are much smaller. The maximum overhead is 40% (*matmult*), and for four applications it is at least 26%.

The two caches most competitive with the adjustable cache are $Fixed(16)$ and $Prefetch(16,4)$. $Fixed(16)$ introduces 25% overhead for *kmerge* and 20% overhead for *bsort*, but is better than $Vblock(8,64,64,(1,1))$ on two programs with fine-grain sharing by 9% and 6%. $Prefetch(16,4)$ reduces the overhead on programs with good spatial locality down to 20% (*kmerge*) and 16% (*bsort*), while outperforming $Vblock(8,64,64,(1,1))$ on three programs by 9%, 3% and 1%.

As noted earlier we optimized the adjustable cache for programs with good spatial locality. That means we provide the optimal *Fixed* cache performance for those programs at the expense of relatively weaker performance for applications with fine-grain sharing. If a 20% performance hit is acceptable for programs with good locality we can use $Prefetch(16,4)$; otherwise we should use $Vblock(8,64,64,(1,1))$. For applications without fine-grain sharing both caches provide more or less equal performance, with $Vblock(8,64,64,(1,1))$ winning for six applications by up to 14% and $Prefetch(16,4)$ winning for three applications by up to 9%.

7.4 Low Bandwidth Machines

In this section we discuss the mean cost per reference for a machine with low bandwidth ($F_{bandwidth} = 20$). The range of cache line sizes for fixed line size caches preferred by our applications is the widest for low bandwidth machines (table 4.2) and includes cache lines between 4 and 64 words.

The lower the bandwidth, the wider the range, and the more applications prefer shorter lines over longer lines. Therefore the adjustable cache we selected for these machines, $Vblock(4,64,16,(1,1))$, has a relatively small initial block size, equal to 16 words, and a small minimum block size of 4 words. The selection of the initial block size is a compromise between applications with fine-grain sharing and applications with good spatial locality. If we selected an initial block size of 64 words as in the previous machines, applications with fine-grain sharing would perform very badly because of the high cost of data transfers. For long running programs splitting would eventually adjust the block size downward for fine-grain sharing. However, in our case all programs which preferred small blocks were quite short, so the effect of the initial block size is very important.

Table 7.4: $M(50,20)$ Machine - *MCPR* Relative to $Vblock(4,64,16,(1,1))$

Application	Cache						
	Fxd(4)	Fxd(8)	Fxd(16)	Fxd(32)	Fxd(64)	Prftch(8,8)	Prftch(16,4)
matmult	1.12	1.11	1.29	2.07	4.09	1.18	1.41
pgauss	1.06	1.19	1.53	2.71	5.27	1.19	1.55
qsort	1.02	1.07	1.35	2.13	4.72	1.08	1.41
plytrace	1.01	0.99	1.16	1.65	3.09	1.10	1.29
pmatmult	1.23	1.12	1.12	1.28	1.74	1.09	1.11
sorbyc	0.99	1.00	1.13	1.41	2.01	1.00	1.13
gauss	1.08	1.08	1.08	1.17	1.41	1.06	1.08
mp3d	1.02	1.07	1.40	2.19	4.37	1.28	1.61
bsort	1.64	1.25	1.05	0.95	0.91	1.16	1.02
kmerge	1.62	1.23	1.03	0.94	0.92	1.17	1.01
sorbyr	1.30	1.13	1.05	1.01	1.00	1.12	1.05

Table 7.4 gives the mean cost per reference of all applications on the $M(50,20)$ machine. The *MCPR*, given for five *Fixed* and two prefetching caches, is relative to $Vblock(4,64,16,(1,1))$. The adjustable cache outperformed the other caches in most of the cases, because only very few numbers are less than 1.

For programs with good spatial locality, like *bsort* and *kmerge*, the two fixed line size caches with the longest lines outperform $Vblock(4,64,16,(1,1))$, but not by more than 9% ($Fixed(64)$). On the other hand, caches with long lines like $Fixed(32)$ and $Fixed(64)$ perform very poorly for applications with fine-grain sharing, as expected. Compared with $Vblock(4,64,16,(1,1))$, they introduce at least 300% overhead in *MCPR* for $Fixed(64)$ and at least 100% overhead for $Fixed(32)$ on programs like *matmult*, *pgauss*, and *qsort*.

$Fixed(16)$ and $Prefetch(16,4)$ are always inferior to $Vblock(4,64,16,(1,1))$ for this machine, and introduce substantial overhead (up to 50%) for programs with fine-grain sharing.

On one out of eleven applications $Fixed(4)$ outperforms the adjustable cache, and the improvement in *MCPR* is negligible (1%). For ten out of eleven applications, $Fixed(4)$ is worse than the adjustable cache, and for both *kmerge* and *bsort* the overhead in *MCPR* is at least 60%.

The two caches most competitive with the adjustable cache are $Fixed(8)$ and $Prefetch(8,8)$. However, $Fixed(8)$ was better than the adjustable cache only once, and only by 1%. And $Fixed(8)$ performed poorly for *bsort*, introducing 25% overhead in *MCPR*, and *kmerge* (23% overhead). For six out of eleven applications, $Fixed(8)$ was worse by more than 10%. $Prefetch(8,8)$ is always worse than the adjustable cache, and for seven out of eleven applications it is worse by 10% or more. This prefetching cache improves a little over $Fixed(8)$ for programs with good spatial locality, but is worse than $Fixed(8)$ for

programs with fine-grain sharing. The maximum overhead in *MCPR* over the adjustable cache is 28% (*mp3d*).

Overall, for low bandwidth machines, adjustable caches improve the mean cost per reference more than for high bandwidth machines. This is because the range of line sizes preferred by applications is wider for low bandwidth machines. Moreover the cost of transferring unneeded data is higher for low bandwidth machines, and adjustable caches are effective in cutting the amount of data transferred.

7.5 High Latency Machines

As we noted in section 4.3, increases in network latency increase the penalty of a mismatch between an application's sharing pattern and the cache line size. Therefore we would expect that the benefits of adjustable caches increase with increased latency. This should be true as long as the dominant factor in the reduction of *MCPR* by an adjustable cache is due to a reduction in the miss rate rather than *DTPR*, because longer latencies increase the impact of the miss rate on *MCPR*.

To verify the impact of latency we ran our simulations for machines with latencies set to 100 and 200 cycles. We did not try to select the best possible adjustable cache for each machine, but instead used one reasonable instance of the adjustable cache for all these experiments, namely $Vblock(16,64,64,(1,1))$. This cache was also used to study high bandwidth machines with latency set to 50 cycles.

Table 7.5: Utility of Adjustable Cache as a Function of Latency and Bandwidth. This table gives the maximal increase in *MCPR* for the best per-machine fixed cache line size when compared with $Vblock(16,64,64,(1,1))$. For each machine we define the best fixed line size cache as the cache that minimizes this increase.

$F_{bandwidth}$	$F_{latency}$		
	50	100	200
0 (infinite)	15%	20%	31%
1 (400 MB/sec)	18%	22%	31%
5 (80 MB/sec)	25%	39%	40%

Table 7.5 illustrates how much we lose if we select the best fixed line size cache instead of $Vblock(16,64,64,(1,1))$. For each machine, we define the best fixed line size cache as the cache that minimizes the maximal relative difference between its performance and $Vblock(16,64,64,(1,1))$ across our application suite. We consider only high bandwidth machines, because high latency machines are likely to have high network bandwidth. We see that the benefits of $Vblock(16,64,64,(1,1))$ increase with an increase in latency, and decrease with an increase in bandwidth, which confirms the results presented earlier in this chapter. The results for unlimited bandwidth machines are very close to those

machines with 400 MB/sec bandwidth, which suggests that increased latency makes adjustable caches more useful regardless of the bandwidth.

Table 7.6: *MCPR* of Best Fixed Line Size Cache Relative to $Vblock(16,64,64,(1,1))$.

Application	BestFixed(M(50,1))	BestFixed(M(100,1))	BestFixed(M(200,1))
matmult	1.15	1.22	1.29
pgauss	1.18	1.19	1.20
qsort	1.14	1.13	1.13
plytrace	1.04	1.05	1.06
pmatmult	1.04	1.06	1.09
sorbyc	1.02	1.02	1.02
gauss	1.00	1.00	1.00
mp3d	1.01	0.97	0.96
bsort	1.08	1.15	1.26
kmerge	1.10	1.18	1.31
sorbyr	1.02	1.04	1.08

It is interesting to look closer at the results for the 400 MB/sec machines as it is most likely that big machines with high latencies will also use a high-bandwidth interconnection network. Table 7.6 gives the performance of the best fixed line size cache relative to $Vblock(16,64,64,(1,1))$ for three machines with bandwidth set to 400 MB/sec and latencies of 50, 100 and 200 cycles. Interestingly, for all these machines, the best fixed line size cache is $Fixed(32)$. Increased latency does not change the best fixed line size cache because our application suite contains a number of programs with fine-grain sharing.

For all applications except *mp3d*, $Vblock(16,64,64,(1,1))$ outperformed the best fixed line size cache for each machine. In the case of *mp3d*, the adjustable cache came within 4% of the best fixed line size cache. We can see that in nearly all cases the higher the latency the better the adjustable cache looks compared to the best fixed line size cache. When latency is 50 cycles, we have four applications for which the best fixed line size cache is worse by 10%-18%; when latency is 100 cycles, the best fixed line size cache is worse by 15%-22% for four applications, and with latency of 200 cycles this range increases to 20%-31%. This trend is due to the lower miss rate of adjustable caches compared to $Fixed(32)$ (as seen in the miss rate figures in appendix A).

In short, this data indicates that an increase in network latency, whether due to improvements in processor speeds or the longer communication paths required in large-scale machines, will magnify the performance benefits of adjustable block size caches seen here, and render fixed line size caches even less competitive.

7.6 Summary

The selection of an adjustable cache for a given multiprocessor depends on the expected workload and the characterization of the host machine. If we expect to run programs which uniformly prefer long cache lines or which only exhibit fine-grain sharing, then we do not need an adjustable cache. On the other hand, if our workload is not uniform with respect to sharing behavior, and different programs prefer different cache line sizes, then an adjustable cache can be used to make our programs perform well. We can get even more performance improvement if, within one program, different data prefer different block sizes.

The most important knowledge needed for the selection of a particular instance of the adjustable cache organization for a given machine is the range of cache line sizes preferred by our workload on that machine and the distribution of applications within this range. This range determines the minimum and the maximum block size for our adjustable cache. Both the merge and split thresholds should be low to ensure good response to the sharing behavior exhibited by an application. For the same reason the size of the split-merge counter should be small. The initial block size depends on the distribution of the applications within the range of preferred (fixed size) cache lines and on the fact that splitting is much easier than merging. Usually the initial block size can be larger than the most popular optimal cache line size, especially for higher bandwidth machines. A larger initial block size makes programs with good locality perform better and allows for fetches of private data with large blocks. In such cases a low split threshold is also necessary to ensure fast convergence to the optimal block size for shared data.

For each machine we were able to select an adjustable cache that outperformed all of the fixed line size caches for at least nine out of eleven applications. For infinite or very high bandwidth machines (with latency of 50 cycles) the best fixed line size caches introduced overhead between 16% and 18% for at least one application. When latency was doubled to 100 cycles this overhead shot up to 30%, and with latency of 200 cycles it was 44%. On medium or low bandwidth machines (with latency of 50 cycles) the overhead incurred by every fixed line size cache was between 21% and 29% for at least one application.

We conclude that the utility of adjustable caches depends on the latency and bandwidth of the interconnection network of the host multiprocessor. Higher bandwidth makes adjustable caches less useful, whereas increased latency has the opposite effect. The impact of increased latency is much stronger than increased bandwidth, and even with infinite bandwidth, there are applications that benefit substantially from adjustable caches.

8 Conclusions and Future Work

In this dissertation we investigated the effects of block size on the performance of coherent caches in shared-memory multiprocessors. Our experiments confirmed that the size of coherency blocks can have a significant impact on the performance of parallel applications, and that there is no single block size that satisfies all applications. We discovered that a small number of block sizes suffices, even if we consider applications with very different sharing patterns, and a range of multiprocessors with wide variations in network latency and bandwidth. Exploiting this fact, we proposed a new cache organization that varies the block size over a small range of sizes according to recent reference behavior. We showed that by dynamically adjusting the block size, the new cache organization provides performance comparable to the best fixed line size cache for each individual application.

In experiments evaluating fixed line size caches, we found that there are applications with good spatial locality which have a strong preference for long cache lines, independent of the network characteristics of the host multiprocessor. However, the performance improvements offered by increasing the cache line size diminish rapidly, and are negligible beyond a certain point (as determined by the latency). In particular, when latency is 50 cycles, there is no benefit to cache lines longer than 64 words; however, when latency is 100 cycles, there are applications that can benefit from cache lines of 128 words.

For applications lacking good spatial locality, the optimal cache line size depends on the product of network bandwidth and latency. An increase in latency or bandwidth favors longer cache lines, while limited bandwidth or low latency favors shorter cache lines. Nonetheless, in the absence of synchronization references, there is no need for cache lines shorter than 4 words.

The range of line sizes favored by programs in our application suite moves upward with an increase in latency. This range also moves upward with an initial increase in bandwidth, but eventually reaches a point at which additional bandwidth has no effect. The higher the latency, the sooner increases in bandwidth no longer affect the range of preferred line sizes. Regardless of the latency, additional bandwidth beyond 400 MB/sec has little effect on the performance of the best fixed line size cache.

Since our applications prefer very different line sizes, there is always a performance penalty if we select one fixed line size cache for each machine. To estimate this penalty

for a given fixed line size cache, machine, and set of applications we considered the maximum relative difference between the performance of that cache for each application and the performance of a fixed line size cache whose line size is best suited to that program. The line size that minimizes this difference is an optimal selection for a given machine and a given application set. In our experiments this minimal relative difference in performance was substantial, ranging from 36% to 68% depending on the latency. Even on a machine with infinite bandwidth, the performance penalty ranges between 11% and 37%. We observed that this performance penalty increases with the latency, and decreases with an increase in bandwidth, although the effect of latency is greater.

To satisfy the needs of multiple applications, spanning various degrees of spatial locality and granularity of sharing, we proposed a new cache organization, called adjustable block size coherent caches. In comparing adjustable block size caches with fixed line size caches (with and without prefetching), we found that the adjustable block size caches offer better performance in most cases. The performance improvements are due to a reduction in the miss rate, while keeping the amount of data transferred low. The miss rate is reduced in two ways: by splitting long cache blocks that are referenced from different processors we avoid false sharing, and by merging short cache blocks used by the same processor we exploit spatial locality. The amount of data transferred is also reduced when we split cache blocks to eliminate false sharing, since in this case, a reduction in the number of transfers contributes to a reduction in the amount of data transferred per reference (*DTPR*).

In some cases the adjustable block size cache managed to reduce the miss rate by up to 30% below the lowest miss rate achieved by any fixed line size cache. This happens when the application prefers more than one block size, i.e., when there is both fine-grain sharing and good spatial locality. In other cases the miss rate produced by the adjustable block size cache was comparable to the lowest miss rate achieved by a fixed line size cache.

The amount of data transferred by the adjustable cache was never below the best *DTPR* achievable by a fixed line size cache, because some fixed line size caches use very short lines, even though short lines result in a very high miss rate. For those fixed line size caches with reasonable miss rates, the adjustable cache usually produced a lower *DTPR*.

Over a range of machines, representing a wide variation in network latency and bandwidth, the adjustable cache was able to out-perform the most competitive fixed line size cache on nine out of eleven applications, and come within 9% of it for the other two applications. The amount of improvement introduced by an adjustable cache depends on the bandwidth and latency of the host machine. For very high bandwidth machines (400 MB/sec or more), the best fixed line size caches introduce overhead between 16% and 18% for at least one application when latency is 50 cycles. When latency is 100 cycles, this overhead rises to 30%, and when latency is 200 cycles, the overhead is 44%. On medium or low bandwidth machines, the overhead is between 21% and 29% for at least one application when latency is 50 cycles.

With respect to cache size, the *min-block* implementation of adjustable caches introduces manageable space overhead over the most space-efficient fixed line size cache.

For the three adjustable caches we considered, this overhead was no higher than 46%.

The choice of parameters for an adjustable cache for a given multiprocessor depends on the expected workload and the network characteristics of the host machine. The most important knowledge needed for the selection of a particular instance of the adjustable cache organization for a given machine is the range of cache line sizes preferred by the workload, and the distribution of applications within this range. This range determines the minimum and maximum block size for the adjustable cache. Both merge and split thresholds should be low enough to ensure good response to the sharing behavior exhibited by an application. For the same reason the size of the split-merge counter should be small. The initial block size depends on the distribution of the applications within the range of preferred (fixed size) cache lines, and on the fact that splitting is much easier than merging.

This dissertation presents the motivation, rationale, and performance implications for adjustable block size caches. The next logical step is to build a multiprocessor that incorporates this cache organization. There are several problems to resolve: the additional complications to the logic of the processor cache, the design of a tag cache to implement the directory, and the complications to the coherency protocol to support multiple block sizes. A successful hardware implementation would provide the ultimate validation of the ideas in this thesis. It is possible to validate the concepts in this thesis without significant hardware modifications however. Many of these ideas apply equally well in an operating system implementation of distributed shared memory (see section 2.1.3) with multiple page sizes.

The implementation of a distributed shared memory with adjustable page sizes would not only provide arguments in favor of adjustable block size coherent caches, but would also constitute a legitimate research project that could increase the viability of distributed shared memory. Currently, the large page sizes used for virtual memory make the false sharing problem even more important for DSM systems than for coherent caches. Small virtual memory pages would address the problem of false sharing, but at the expense of a significant increase in the number of TLB misses. Multiple page sizes seem like an ideal solution for DSM, provided we have an algorithm for deciding which page size to use for each data object. Fortunately, we can use exactly the same algorithm we used to adjust the block size in coherent caches! That is, with each variable size page we could associate size bits, a split-merge counter, two reference bits, and state bits, and then split and merge pages on a coherency miss according to the recent reference stream to a page.

In order to implement this idea, we would require the virtual memory hardware to support multiple page sizes. Such hardware already exists. For example, the TLB in the R4000 microprocessor already has size bits in each TLB entry indicating the size of the page. (Unfortunately the smallest page supported, 4KB, is still too big for our purposes.) We would still require a hardware modification to support two reference bits per TLB entry, although these bits could be simulated in the operating system software (which would of course slow down the distributed shared memory system). All other control fields associated with a block in the adjustable cache organization, like the split-merge counter and page state, can and should be handled by the operating system, as

they are referenced and changed quite rarely.

· In summary this dissertation quantifies the performance penalty associated with a mismatch between an application's sharing pattern and the block size in coherent caches. We proposed a new cache organization that adjusts the size of data blocks dynamically according to recent reference patterns, and thereby reduces this penalty substantially.

Bibliography

[1] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[2] A. Agarwal, B. Lim, D. Krantz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proc. 17th International Symposium on Computer Architecture*, pages 104–114, June 1990.

[3] A. Agarwal, R. Simoni, J.L. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proc. 15th International Symposium on Computer Architecture*, pages 280–289, June 1988.

[4] J. Archibald and J-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4:273–298, 1986.

[5] F. Baskett, T. Jermoluk, and D. Solomon. The 4d-mp graphics superworkstation. In *33rd IEEE Computer Society Intl. Conference*, pages 468–471, 1988.

[6] B.N. Bershad, E.D. Lazowska, and H.M. Levy. Presto: A system for object-oriented parallel programming. *Software - Practice and Experience*, 18(8), August 1988.

[7] R. Bianchini and T.J. LeBlanc. Software caching on cache-coherent multiprocessors. In *Proc. 4th IEEE Symposium on Parallel and Distributed Processing*, pages 521–526, December 1992.

[8] R. Bisiani and M. Ravishankar. Plus: A distributed shared-memory system. In *Proc. 17th International Symposium on Computer Architecture*, pages 115–124, June 1990.

[9] W. J. Bolosky and M. L. Scott. Evaluation of Multiprocessor Memory Systems Using Off-Line Optimal Behavior. *The Journal of Parallel and Distributed Computing*, August 1992. Also URCS Tech. Rpt. 403.

[10] W.J. Bolosky, R.P. Fitzgerald, and M.L. Scott. Simple but effective techniques for NUMA memory management. *Proc. 12th ACM Symposium on Operating System Principles*, pages 19–31, December 1989.

[11] W.J. Bolosky, M.L. Scott, R.P. Fitzgerald, R.J. Fowler, and A.L. Cox. NUMA policies and their relation to memory architecture. *Proc. 4th Intl. Conf. on Architectural Support Support for Prog. Lang. and Oper. Sys.*, pages 212–221, April 1991.

[12] D. Callaham, K. Kennedy, and A. Porterfield. Software prefetching. In *Proc. 4th Intl. Conf. on Architectural Support Support for Prog. Lang. and Oper. Sys.*, pages 40–52, April 1991.

[13] L.M. Censier and P. Feautier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27:1112–1118, December 1978.

[14] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, pages 49–57, June 1990.

[15] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proc. 4th Intl. Conf. on Architectural Support Support for Prog. Lang. and Oper. Sys.*, pages 224–234, April 1991.

[16] H. Cheong and A. Veidenbaum. A cache coherence scheme with fast selective invalidations. In *Proc. 15th International Symposium on Computer Architecture*, pages 299–307, May 1988.

[17] H. Cheong and A.V. Veidenbaum. Compiler-directed cache management in multiprocessors. *IEEE Computer*, pages 39–47, June 1990.

[18] D.R. Cheriton, H.A. Goosen, and P. Machanick. Restructuring a parallel simulation to improve cache behavior in a shared-memory mutiprocessor: A first experience. In *Proc. Intl. Symp. on Shared Memory Multiprocessing*, pages 109–118, April 1991.

[19] D.R. Cheriton, A. Gupta, P.D. Boyle, and H.A. Goosen. The VMP multiprocessor: Initial experience, refinements, and performance evaluation. In *Proc. 15th International Symposium on Computer Architecture*, pages 410–421, June 1988.

[20] A.L. Cox and R.J. Fowler. The implementation of a coherent memory abstraction on a NUMA multiprocessor: Experiences with PLATINUM. *Proc. 12th ACM Symposium on Operating System Principles*, pages 32–44, December 1989.

[21] C. Dubnicki and T.J. LeBlanc. Adjustable block size coherent caches. In *Proc. 19th International Symposium on Computer Architecture*, pages 170–180, May 1992.

[22] M. Dubois and Ch. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Trans. on Software Engineering*, pages 660–673, June 1990.

[23] M. Dubois, Ch. Scheurich, and F.A. Briggs. Memory access buffering in multiprocessors. *Proc. 13th International Symposium on Computer Architecture*, pages 434–442, June 1986.

[24] J. Edler, A. Gottlieb, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, M. Snir, P.J. Teller, and J. Wilson. Issues related to MIMD shared-memory computers: The NYU Ultracomputer approach. *Proc. 12th International Symposium on Computer Architecture*, pages 126–135, June 1985.

[25] S. J. Eggers and T. E. Jeremiassen. Eliminating false sharing. In *Proc. 1991 Intl. Conf. on Parallel Processing*, pages 377–381, 1991. Volume I.

[26] S.J. Eggers. Simplicity versus accuracy in a model of cache coherency overhead. *IEEE Trans. on Computers*, 40(8):893–906, August 1991.

[27] S.J. Eggers. *Simulation analysis of data sharing in shared-memory multiprocessors.* PhD thesis, Computer Science Division (EECS), University of California, April 1989.

[28] S.J. Eggers and R.H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proc. 15th International Symposium on Computer Architecture*, pages 373–383, May 1988.

[29] S.J. Eggers and R.H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proc. 3rd Intl. Conf. on Architectural Support Support for Prog. Lang. and Oper. Sys.*, pages 257–270, April 1989.

[30] S.J. Eggers and R.H. Katz. Evaluation of the performance of four snooping cache coherency protocols. In *Proc. 16th International Symposium on Computer Architecture*, pages 2–15, May 1989.

[31] S.J. Frank. Tightly coupled multiprocessor systems speed memory access times. *Electronics*, 57:164–169, January 1984.

[32] J.R. Goodman, M.K. Vernon, and P.J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. *Proc. 3rd Intl. Conf. on Architectural Support Support for Prog. Lang. and Oper. Sys.*, pages 104–112, Apr 1989.

[33] A. Gupta and W.D. Weber. Analysis of cache invalidation patterns in shared-memory multiprocessors. In *Cache and Interconnect Architectures in Multiprocessors*, pages 83–108. Kulwer Academic Publishers, 1990.

[34] D.V. Jaines, A.T. Laundrie, S. Gjessing, and G. Sohi. Scalable coherent interface. *IEEE Computer*, pages 74–77, June 1990.

[35] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a Cache Consistency Protocol. In *Proc. 12th International Symposium on Computer Architecture*, pages 276–283, 1985.

[36] C.P. Kruskal and M. Snir. The performance of multistage interconnection networks for multiprocessors. *IEEE Trans. on Computers*, 32:1091–1098, 1983.

[37] R. P. LaRowe and C. S. Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, November 1991.

[38] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J.L. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[39] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J.L. Hennessy. The DASH prototype: Implementation and performance. In *Proc. 19th International Symposium on Computer Architecture*, pages 92–103, May 1992.

[40] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *Proc. 5th Annual ACM Symp. on Principles of Distributed Computing*, pages 229–239, August 1986.

[41] Holliday M.A. Reference history, page size, and migration daemons in local/remote architectures. *Proc. 3rd Intl. Conf. on Architectural Support Support for Prog. Lang. and Oper. Sys.*, pages 104–112, April 1989.

[42] J.M. Mellor-Crummey and M.L. Scott. Synchronization without contention. In *Proc. 4th Intl. Conf. on Architectural Support Support for Prog. Lang. and Oper. Sys.*, pages 269–278, April 1991.

[43] B.W. O'Krafka and R.A. Newton. An empirical evaluation of two memory-efficient directory methods. In *Proc. 17th International Symposium on Computer Architecture*, pages 138–147, June 1990.

[44] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proc. 11th International Symposium on Computer Architecture*, pages 348–354, June 1984.

[45] J.P. Singh, W-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Department of Computer Science, Stanford University, April 1991.

[46] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, June 1990.

[47] Charles P. Thacker and Lawrence C. Stewart. Firefly: a multiprocessor workstation. In *2nd Intl. Conf. on Architectural Support Support for Prog. Lang. an d Oper. Sys.*, pages 164–172, Palo Alto, CA, October 1987.

[48] J. Torrellas, M.S. Lam, and J.L. Hennessy. Shared data placement optimizations to reduce multiprocessor cache miss rates. In *Proc. 1990 Intl. Conf. on Parallel Processing Volume II*, pages 266–270, August 1990.

A Figures and Tables for All Applications

For each application we give here the following data:

- miss rate, data transfer rate and cache size. Each of these figures contain data for our 9 base cache instances (five fixed, three adjustable, and one prefetching - see section 6.1),
- for adjustable caches, split and merge statistics giving the distribution of adjustment operations according to block size before each operation,
- for adjustable caches, transfer statistics giving the fraction of transfers according to block size.

A.1 *bsort*

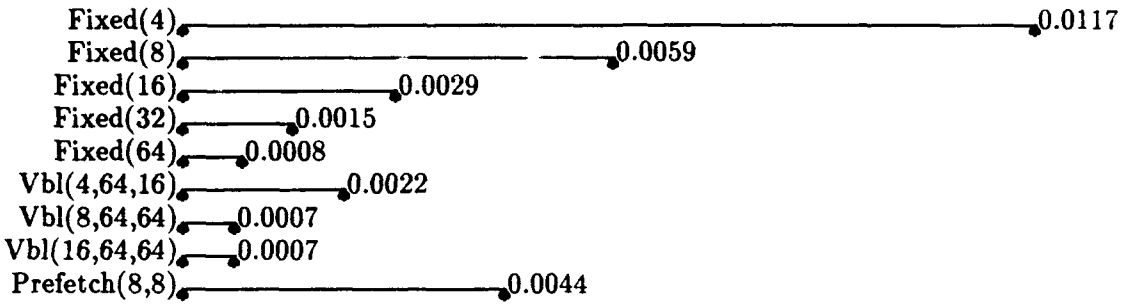


Figure A.1.1: Miss Rate - *bsort*

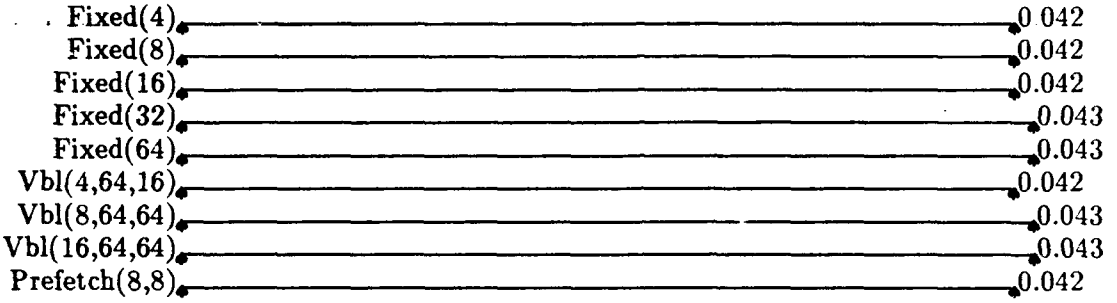


Figure A.1.2: Data Words Transferred Per Reference - *bsort*

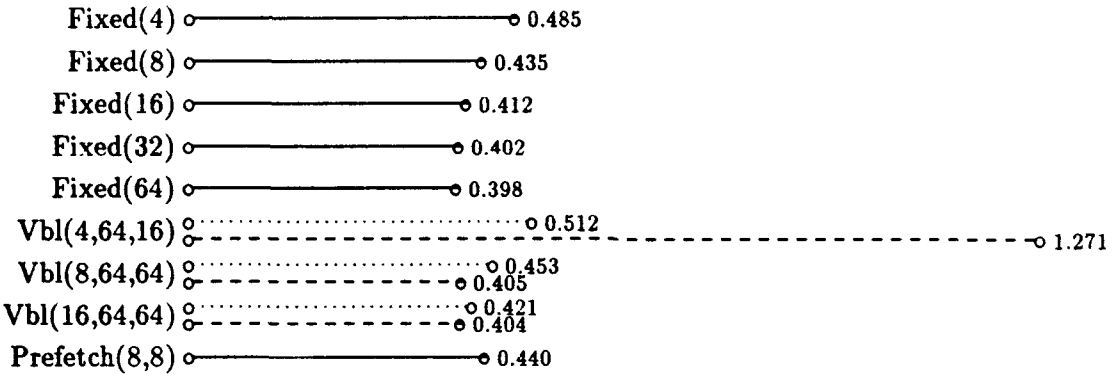


Figure A.1.3: Cache Size (bits per reference) - *bsort*. For adjustable caches dashed line gives the size of the *max-block* implementation, dotted line gives the size of the *min-block* implementation.

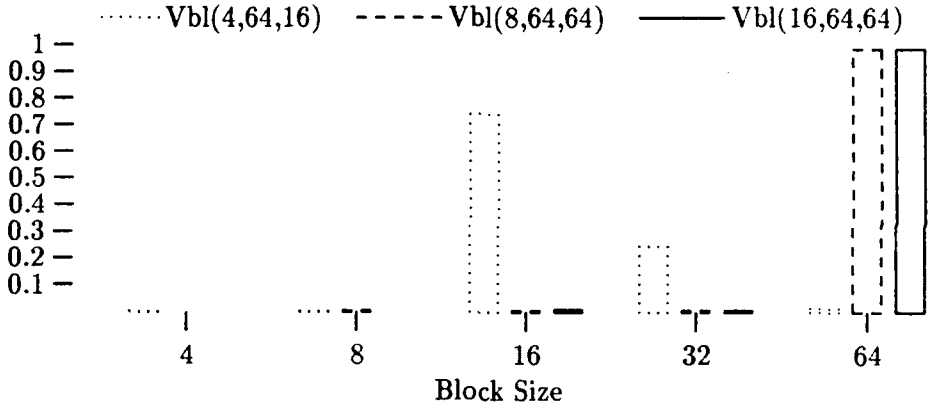


Figure A.1.4: Fraction of Block Transfers of Given Size - *bsort*

Table A.1: Split and Merge Statistics - *bsort*

Cache	Splits					Merges					Failed Merges
	Total	Block Size				Total	Block Size				
		8	16	32	64		4	8	16	32	
Vbl(4,64,16)	53	19	30	4		5442	1	1	5120	320	15432
Vbl(8,64,64)	50		6	17	27	2			1	1	52
Vbl(16,64,64)	44			17	27	2			1	1	42

A.2 *gauss*

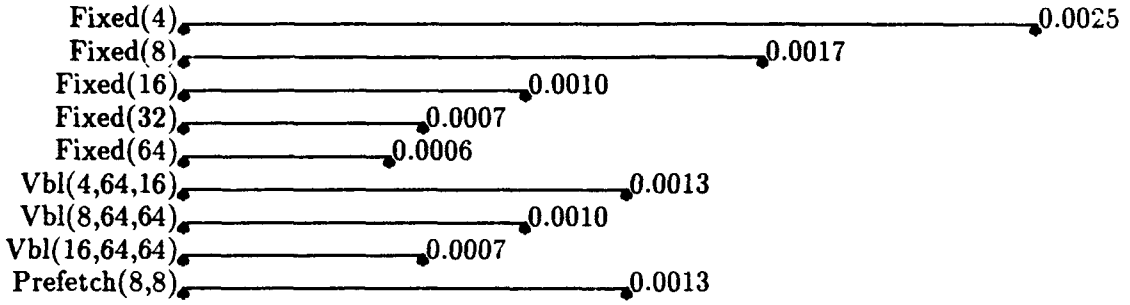


Figure A.2.1: Miss Rate - *gauss*

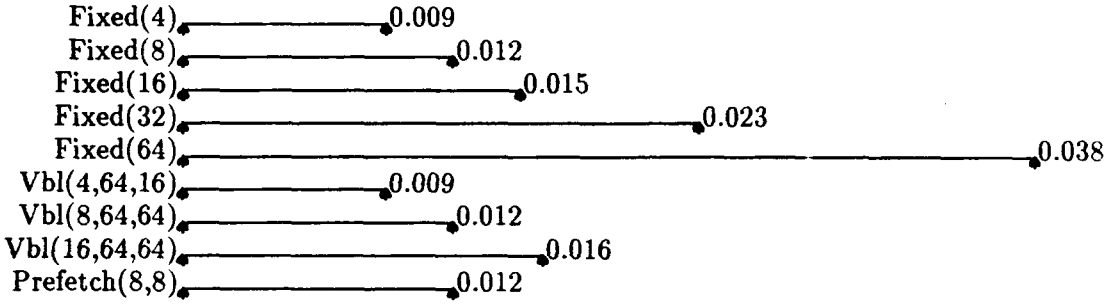


Figure A.2.2: Data Words Transferred Per Reference - *gauss*

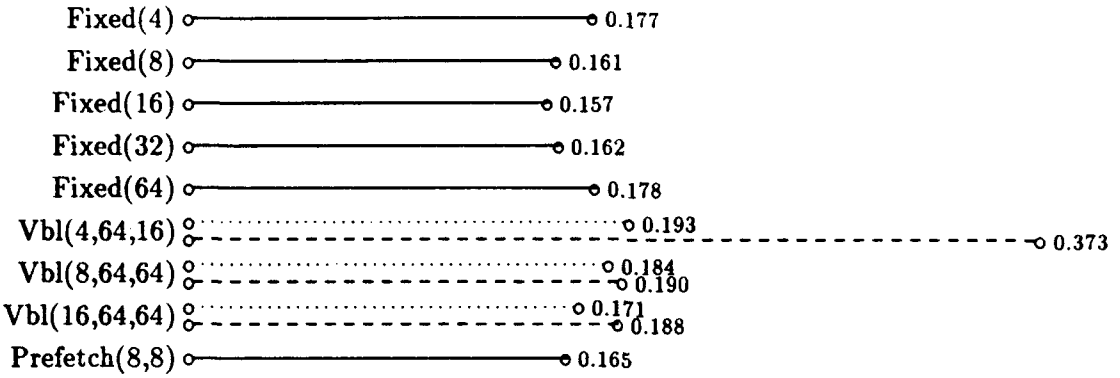


Figure A.2.3: Cache Size (bits per reference) - *gauss*. For adjustable caches dashed line gives the size of the *max-block* implementation, dotted line gives the size of the *min-block* implementation.

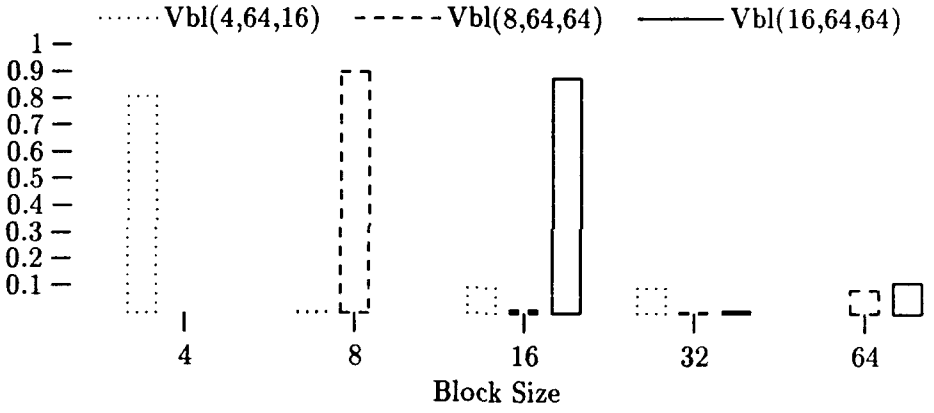


Figure A.2.4: Fraction of Block Transfers of Given Size - *gauss*

Table A.2: Split and Merge Statistics - *gauss*

Cache	Splits					Merges					Failed Merges
	Total	Block Size				Total	Block Size				
		8	16	32	64		4	8	16	32	
Vbl(4,64,16)	1069	505	564			7008			7008		15345
Vbl(8,64,64)	1035		39	461	535	1		1			62
Vbl(16,64,64)	1154			618	536	158			157	1	5026

A.3 *kmerge*

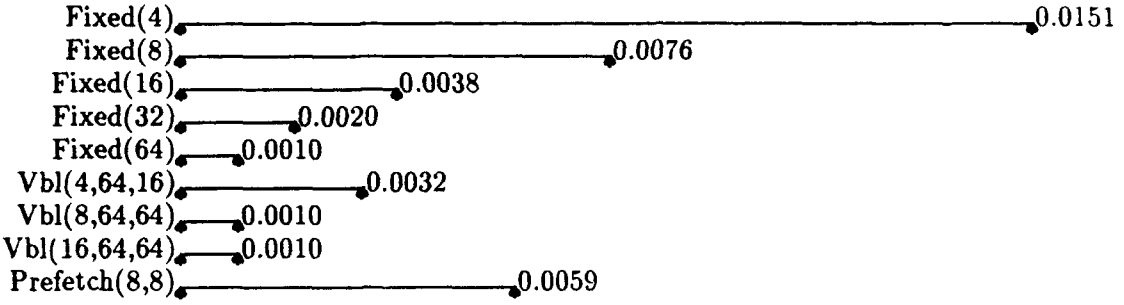


Figure A.3.1: Miss Rate - *kmerge*

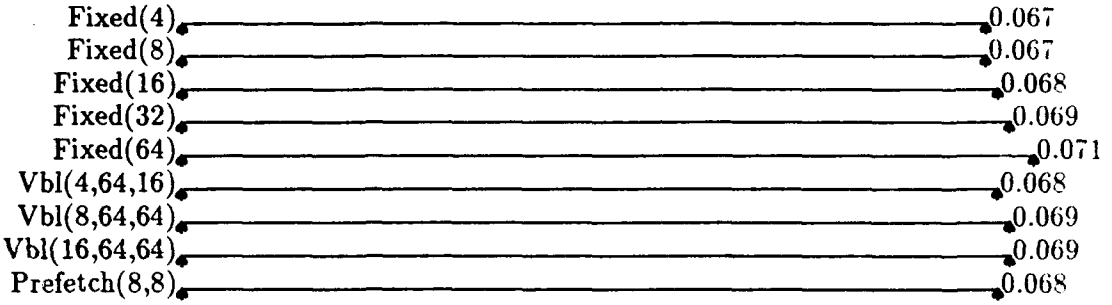


Figure A.3.2: Data Words Transferred Per Reference - *kmerge*

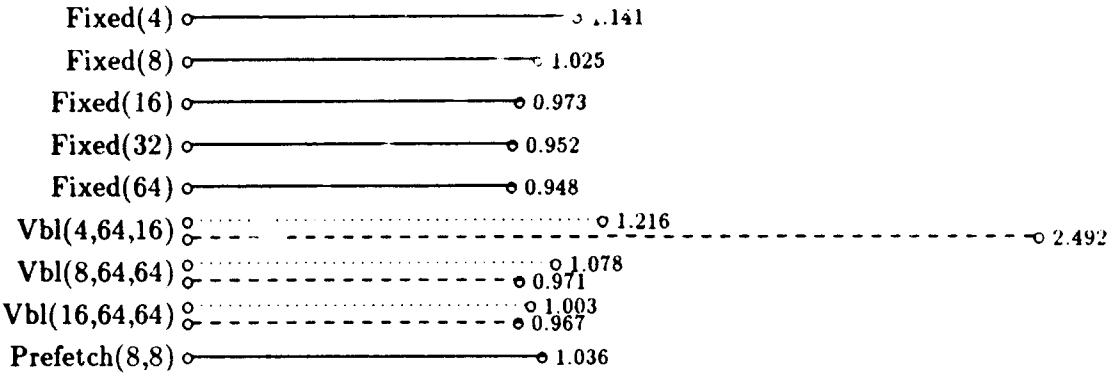


Figure A.3.3: Cache Size (bits per reference) - *kmerge*. For adjustable caches dashed line gives the size of the *max-block* implementation, dotted line gives the size of the *min-block* implementation.

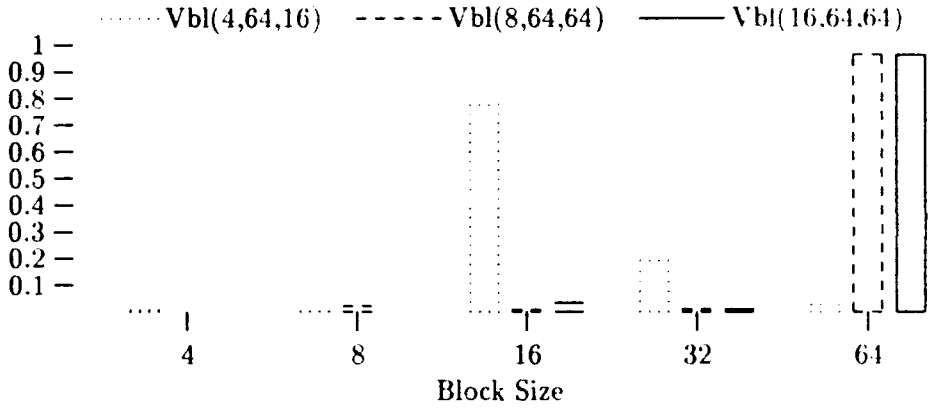


Figure A.3.4: Fraction of Block Transfers of Given Size - *kmerge*

Table A.3: Split and Merge Statistics - *kmerge*

Cache	Splits					Merges					Failed Merges
	Total	Block Size				Total	Block Size				
		8	16	32	64		4	8	16	32	
Vbl(4,64,16)	68	19	38	9	2	4550		1	3990	559	9690
Vbl(8,64,64)	61		10	21	30	4			2	2	38
Vbl(16,64,64)	51			21	30	4			2	2	39

A.4 *matmult*

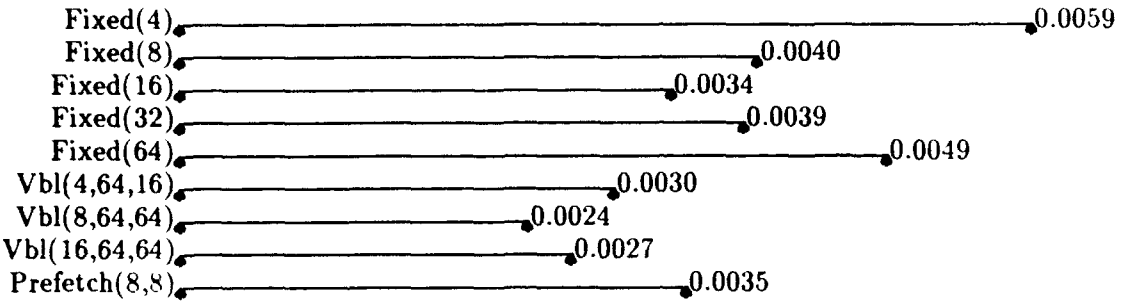


Figure A.4.1: Miss Rate - *matmult*

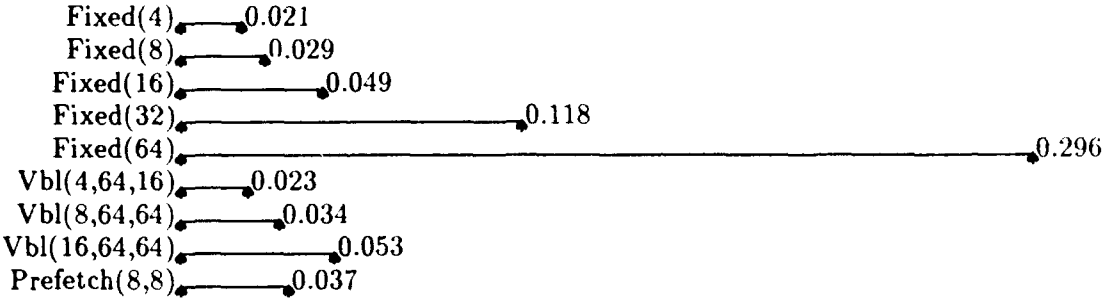


Figure A.4.2: Data Words Transferred Per Reference - *matmult*

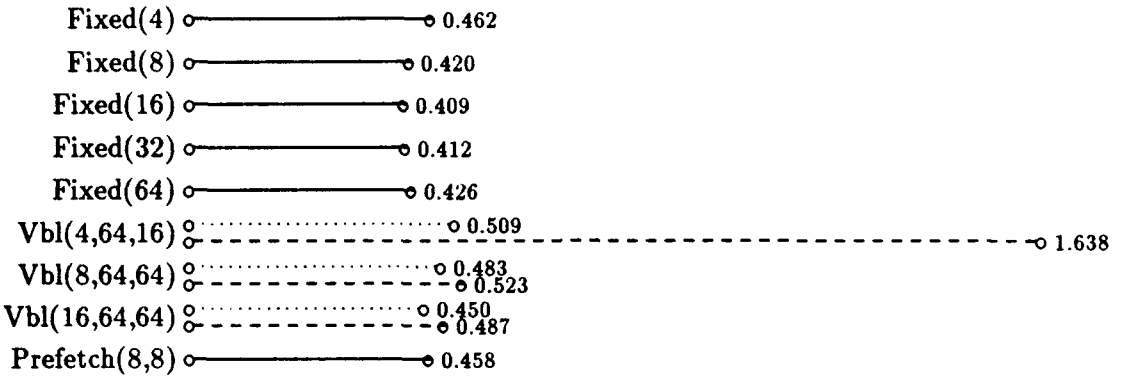


Figure A.4.3: Cache Size (bits per reference) - *matmult*. For adjustable caches dashed line gives the size of the *max-block* implementation, dotted line gives the size of the *min-block* implementation.

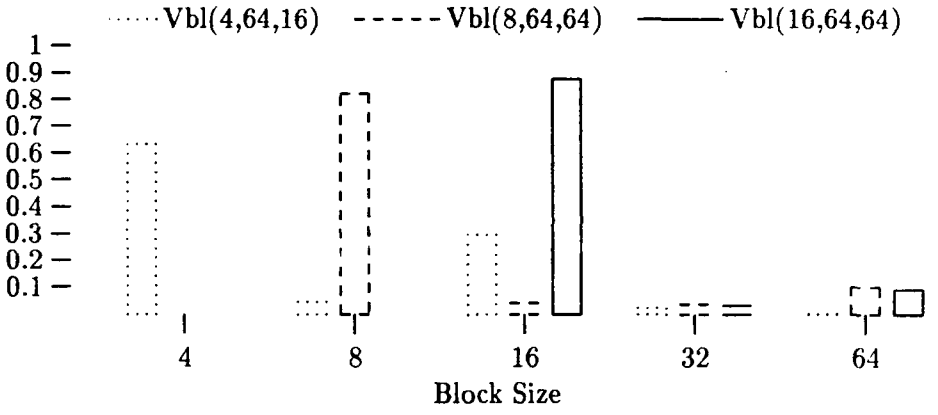


Figure A.4.4: Fraction of Block Transfers of Given Size - *matmult*

Table A.4: Split and Merge Statistics - *matmult*

Cache	Splits					Merges					Failed Merges
	Total	Block Size				Total	Block Size				
		8	16	32	64		4	8	16	32	
Vbl(4,64,16)	290	114	116	56	4	183	29	29	112	13	3309
Vbl(8,64,64)	252		109	79	64	65		37	20	8	805
Vbl(16,64,64)	146			81	65	35			25	10	528

A.5 *mp3d*

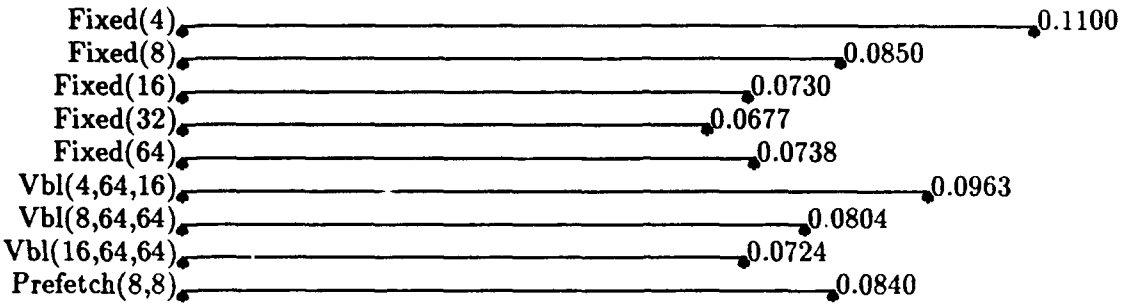


Figure A.5.1: Miss Rate - *mp3d*

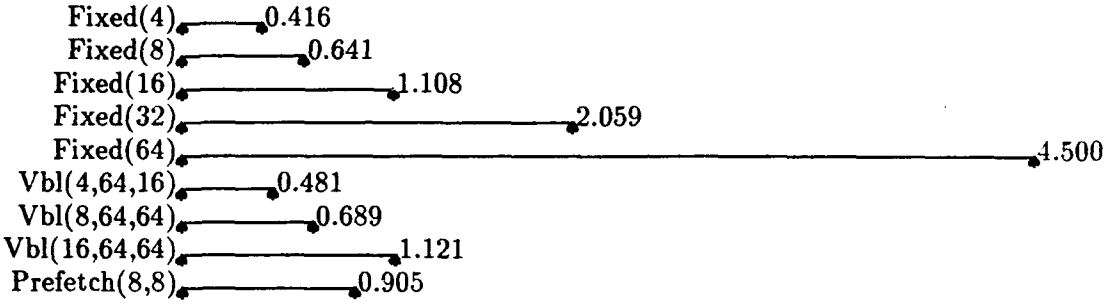


Figure A.5.2: Data Words Transferred Per Reference - *mp3d*

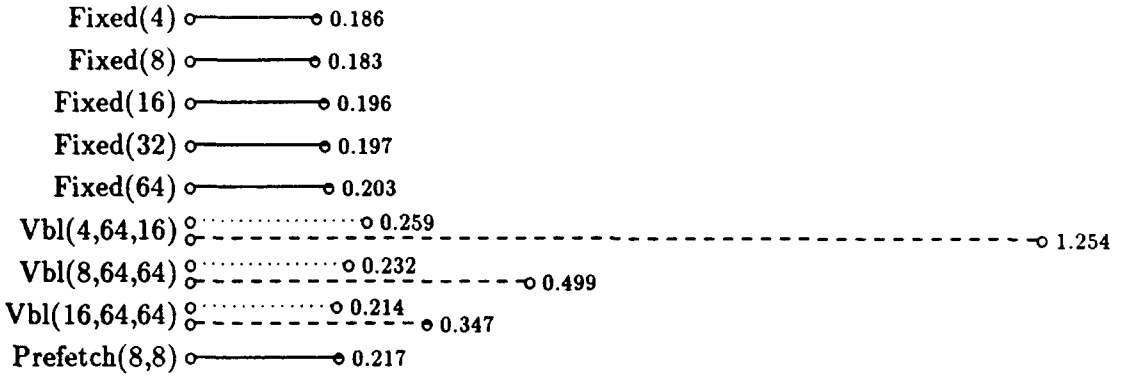


Figure A.5.3: Cache Size (bits per reference) - *mp3d*. For adjustable caches dashed line gives the size of the *max-block* implementation, dotted line gives the size of the *min-block* implementation.

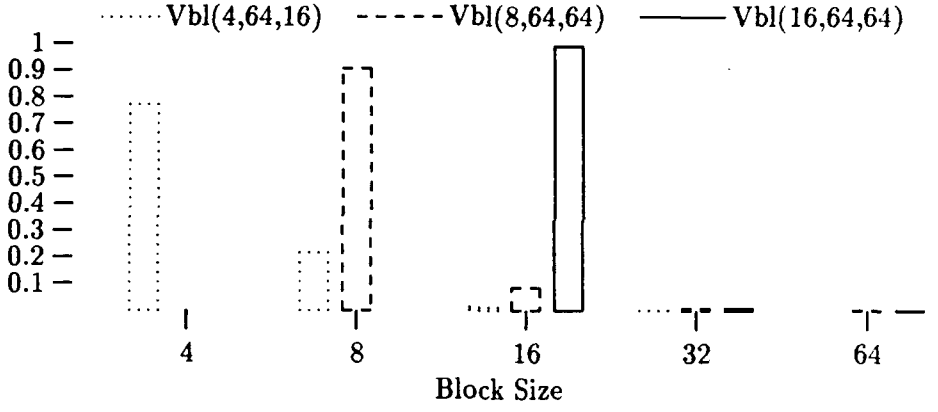


Figure A.5.4: Fraction of Block Transfers of Given Size - *mp3d*

Table A.5: Split and Merge Statistics - *mp3d*

Cache	Splits					Merges					Failed Merges
	Total	Block Size				Total	Block Size				
		8	16	32	64		4	8	16	32	
Vbl(4,64,16)	12816	9625	2937	254		10025	8186	1585	254		593940
Vbl(8,64,64)	20814		17538	2709	567	18508		16462	1942	104	361734
Vbl(16,64,64)	3670			3106	564	2447			2347	100	122035

A.6 *pgauss*

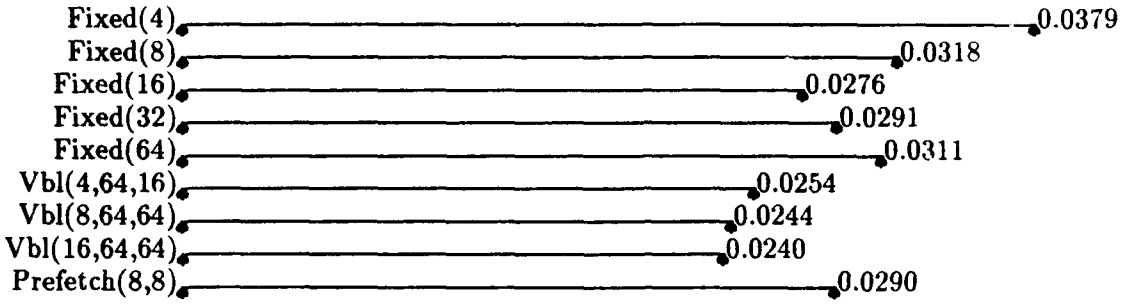


Figure A.6.1: Miss Rate - *pgauss*

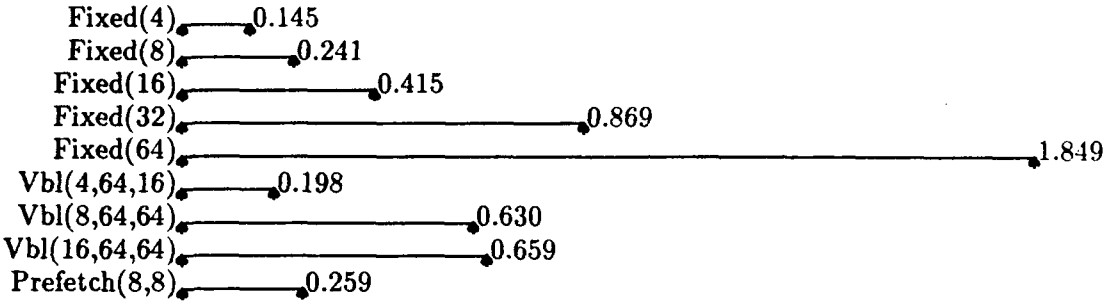


Figure A.6.2: Data Words Transferred Per Reference - *pgauss*

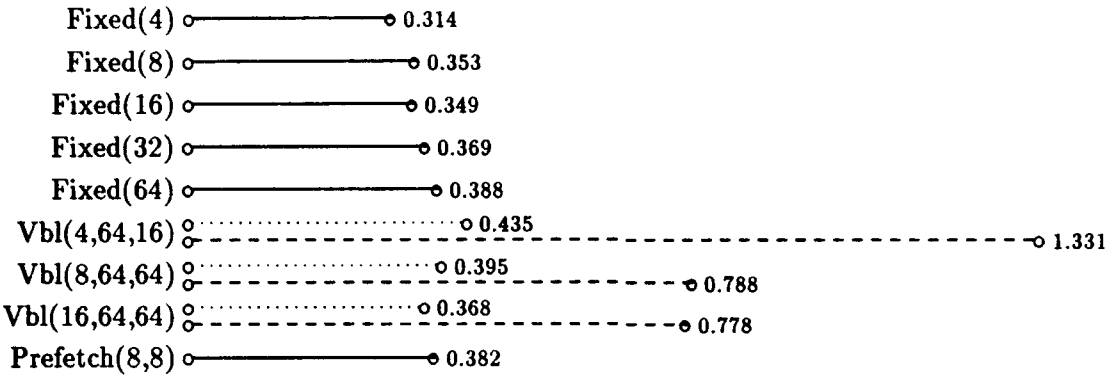


Figure A.6.3: Cache Size (bits per reference) - *pgauss*. For adjustable caches dashed line gives the size of the *max-block* implementation, dotted line gives the size of the *min-block* implementation.

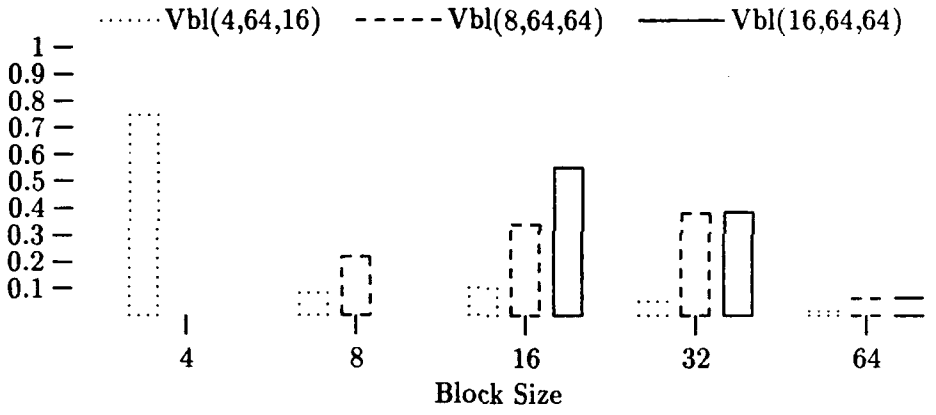


Figure A.6.4: Fraction of Block Transfers of Given Size - *pgauss*

Table A.6: Split and Merge Statistics - *pgauss*

Cache	Splits					Merges					Failed Merges
	Total	Block Size				Total	Block Size				
		8	16	32	64		4	8	16	32	
Vbl(4,64,16)	851	289	435	64	63	429	1		325	103	227285
Vbl(8,64,64)	2837		72	1436	1329	1				1	188092
Vbl(16,64,64)	2765			1436	1329	2			1	1	182610

A.7 *plytrace*

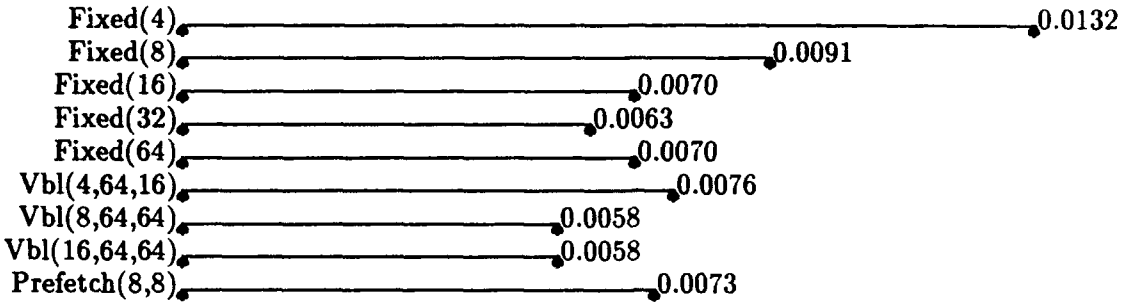


Figure A.7.1: Miss Rate - *plytrace*

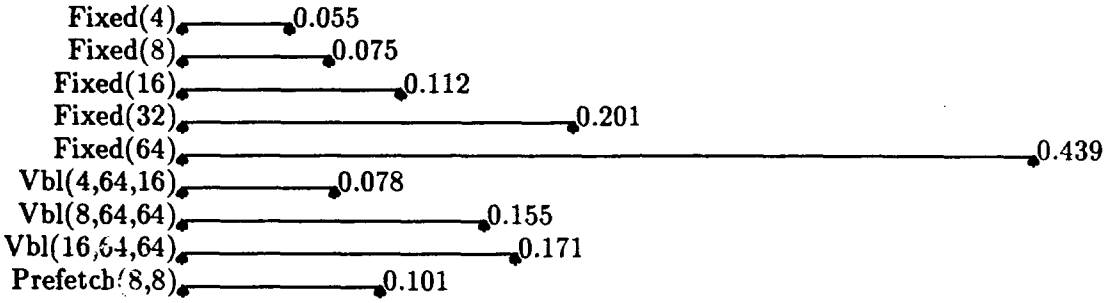


Figure A.7.2: Data Words Transferred Per Reference - *plytrace*

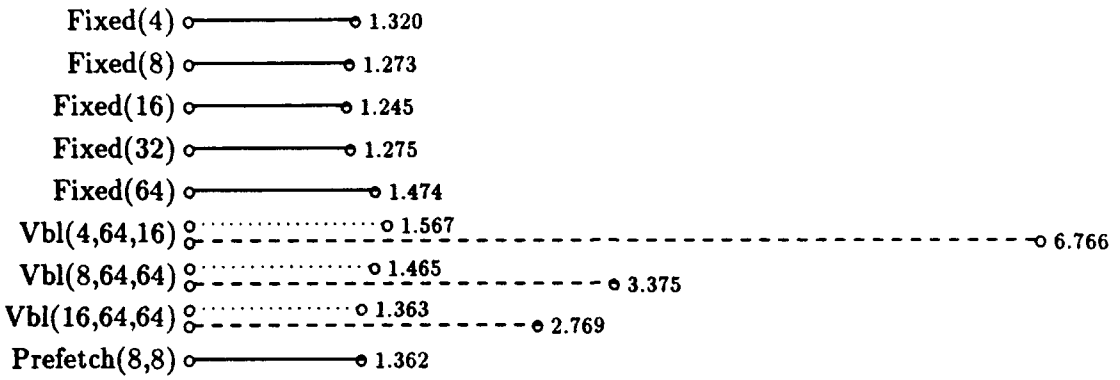


Figure A.7.3: Cache Size (bits per reference) - *plytrace*. For adjustable caches dashed line gives the size of the *max-block* implementation, dotted line gives the size of the *min-block* implementation.

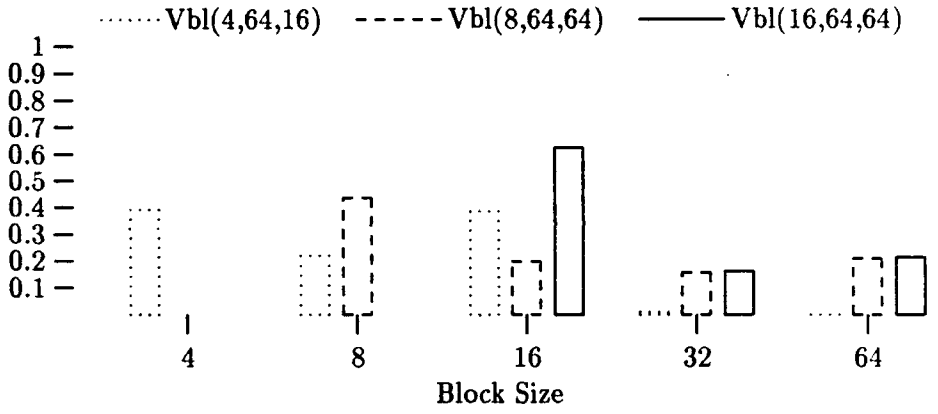


Figure A.7.4: Fraction of Block Transfers of Given Size - *plytrace*

Table A.7: Split and Merge Statistics - *plytrace*

Cache	Splits					Merges					Failed Merges
	Total	Block Size				Total	Block Size				
		8	16	32	64		4	8	16	32	
Vbl(4,64,16)	9688	4047	5513	128		625	30	66	488	41	23261
Vbl(8,64,64)	9681		2623	3881	3177	170		30	87	53	12098
Vbl(16,64,64)	7067			3890	3177	150			97	53	10133

A.8 *pmatmult*

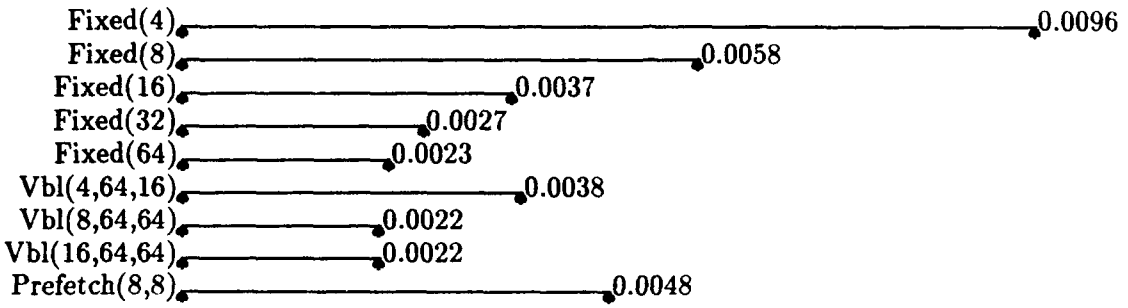


Figure A.8.1: Miss Rate - *pmatmult*

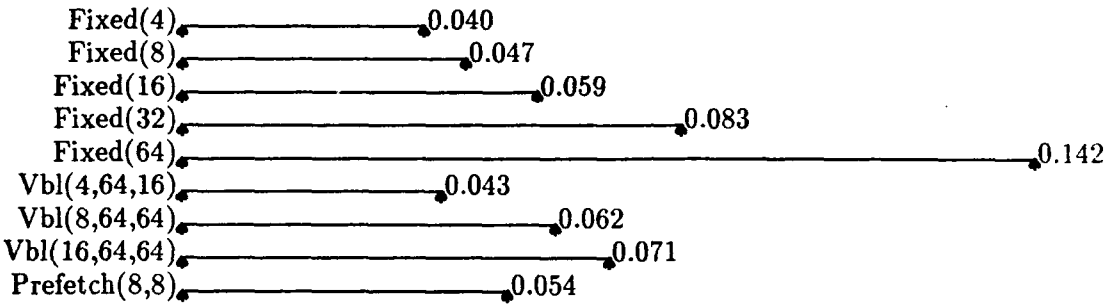


Figure A.8.2: Data Words Transferred Per Reference - *pmatmult*

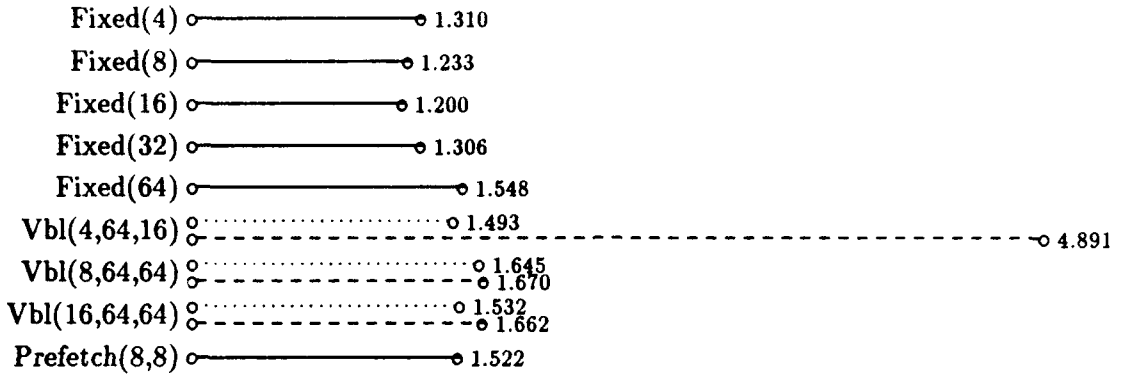


Figure A.8.3: Cache Size (bits per reference) - *pmatmult*. For adjustable caches dashed line gives the size of the *max-block* implementation, dotted line gives the size of the *min-block* implementation.

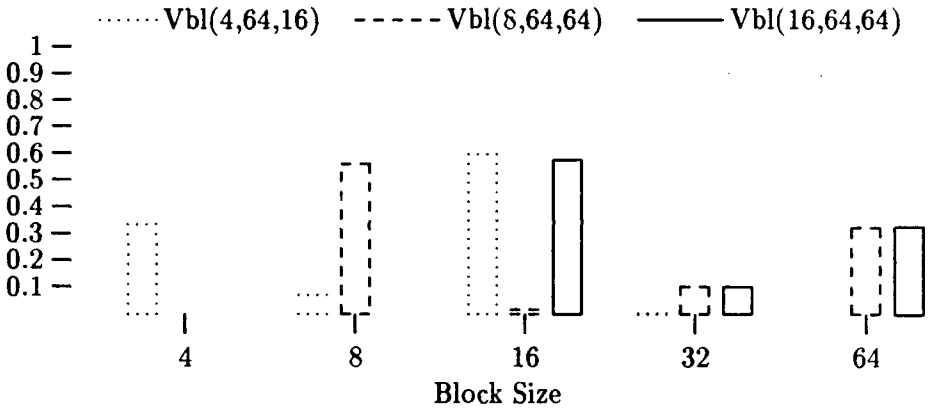


Figure A.8.4: Fraction of Block Transfers of Given Size - *pmatmult*

Table A.8: Split and Merge Statistics - *pmatmult*

Cache	Splits					Merges					Failed Merges
	Total	Block Size				Total	Block Size				
		8	16	32	64		4	8	16	32	
Vbl(4,64,16)	285	27	257	1		17			17		6785
Vbl(8,64,64)	332		13	22	297	1				1	421
Vbl(16,64,64)	319			22	297	1				1	445

A.9 *qsort*

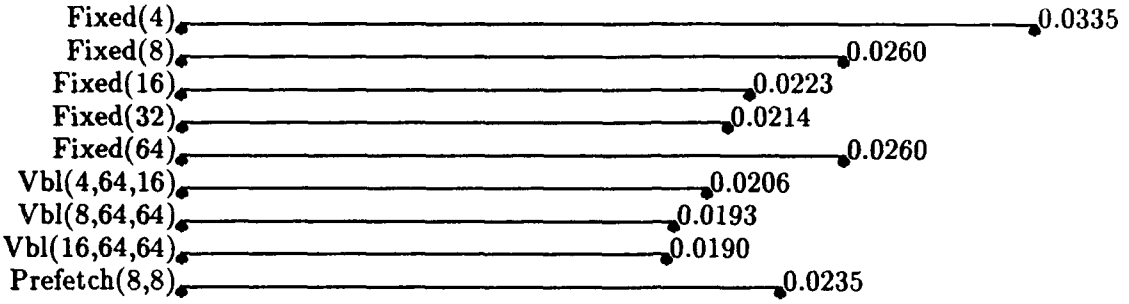


Figure A.9.1: Miss Rate - *qsort*

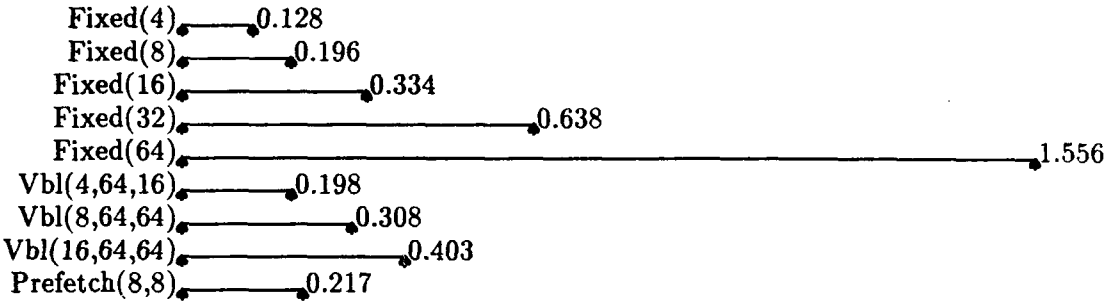


Figure A.9.2: Data Words Transferred Per Reference - *qsort*

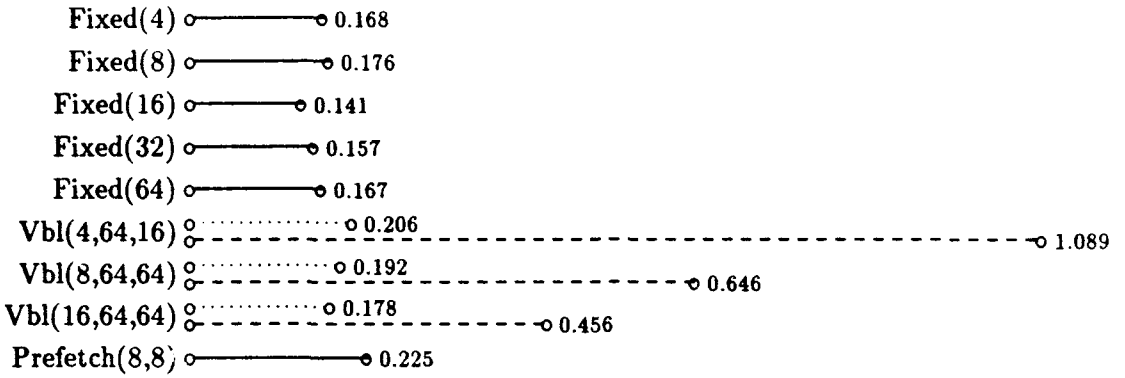


Figure A.9.3: Cache Size (bits per reference) - *qsort*. For adjustable caches dashed line gives the size of the *max-block* implementation, dotted line gives the size of the *min-block* implementation.

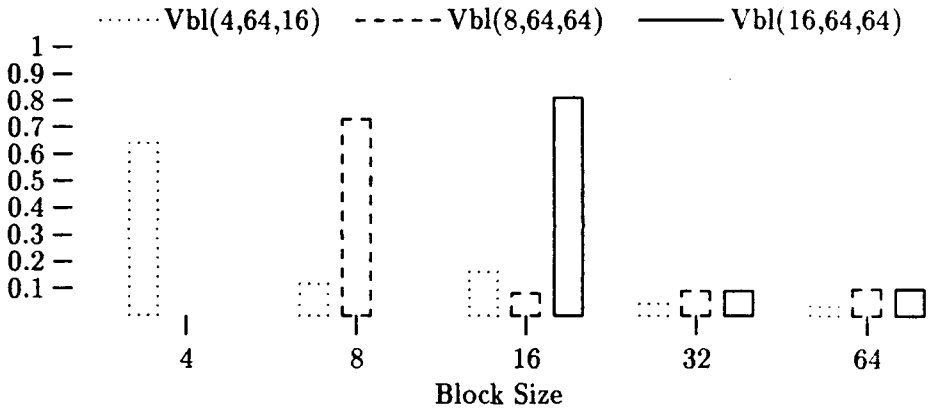


Figure A.9.4: Fraction of Block Transfers of Given Size - *qsort*

Table A.9: Split and Merge Statistics - *qsort*

Cache	Splits					Merges					Failed Merges
	Total	Block Size				Total	Block Size				
		8	16	32	64		4	8	16	32	
Vbl(4,64,16)	3444	886	1390	845	323	1456	17	56	1044	339	131255
Vbl(8,64,64)	2972		1053	1193	726	318		47	250	21	41504
Vbl(16,64,64)	1928			1202	726	285			264	21	36709

A.10 *sorbyc*

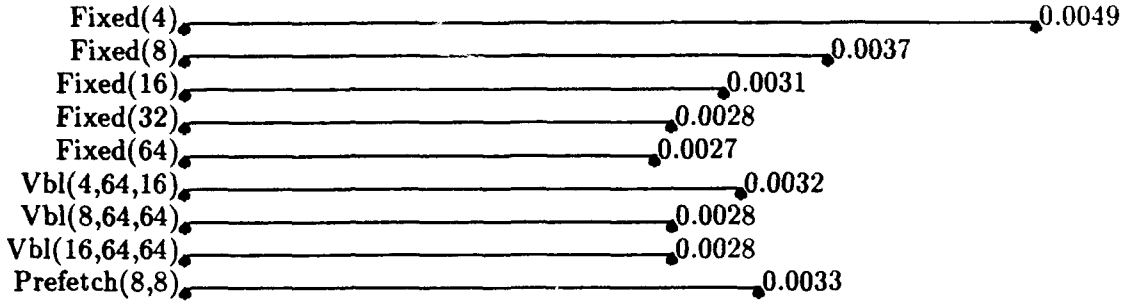


Figure A.10.1: Miss Rate - *sorbyc*

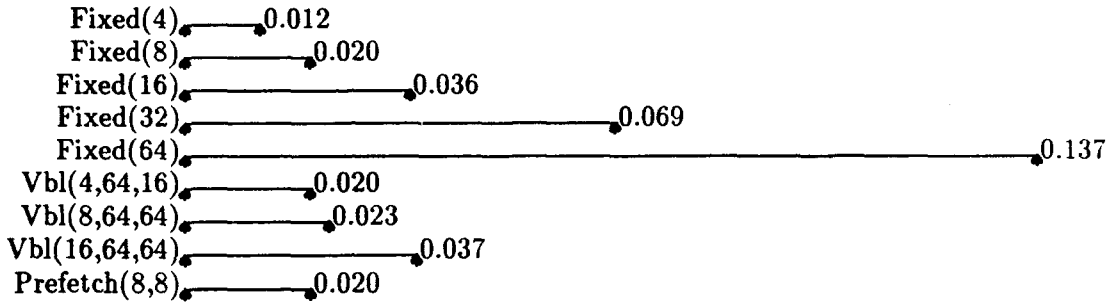


Figure A.10.2: Data Words Transferred Per Reference - *sorbyc*

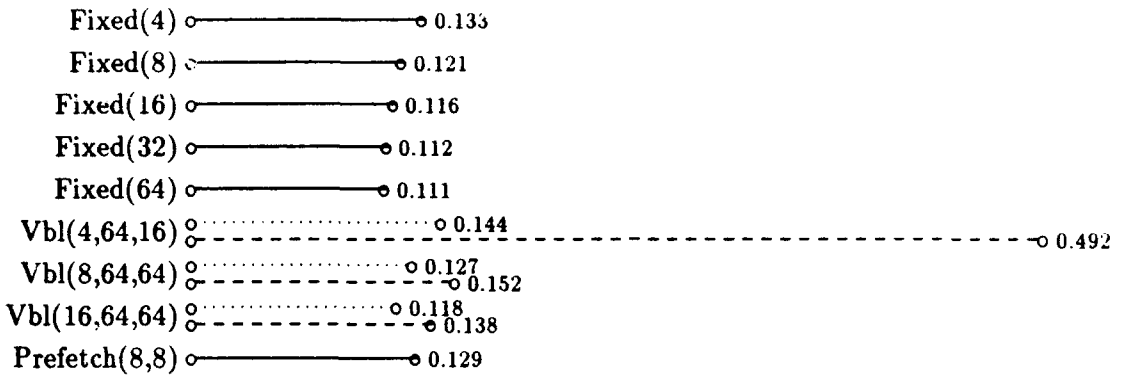


Figure A.10.3: Cache Size (bits per reference) - *sorbyc*. For adjustable caches dashed line gives the size of the *max-block* implementation, dotted line gives the size of the *min-block* implementation.

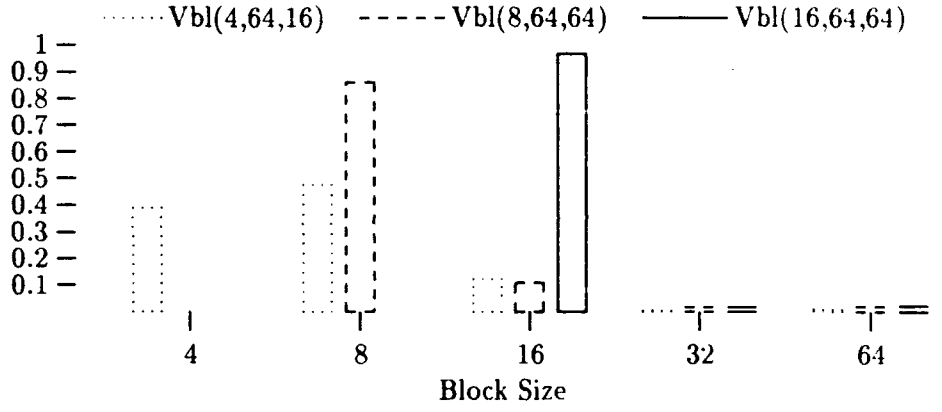


Figure A.10.4: Fraction of Block Transfers of Given Size - *sorbyc*

Table A.10: Split and Merge Statistics - *sorbyc*

Cache	Splits					Merges					Failed Merges
	Total	Block Size				Total	Block Size				
		8	16	32	64		4	8	16	32	
Vbl(4,64,16)	21396	11903	9129	229	135	19605	11567	7617	270	151	59077
Vbl(8,64,64)	11257		9790	507	960	9685		9321	231	133	50133
Vbl(16,64,64)	1436			476	960	333			200	133	1325

A.11 *sorbyr*

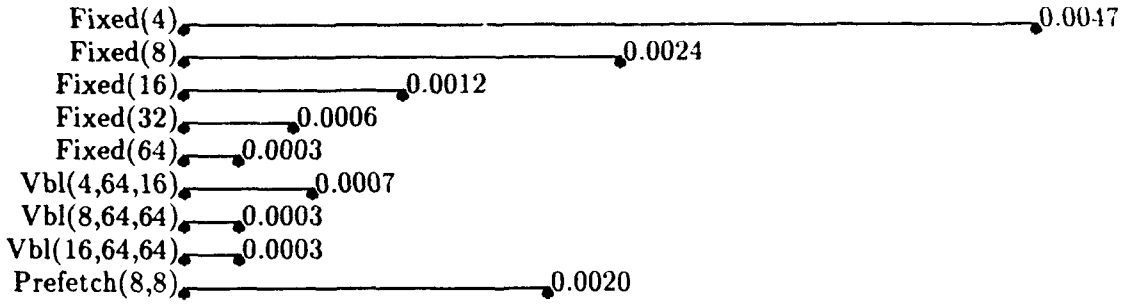


Figure A.11.1: Miss Rate - *sorbyr*

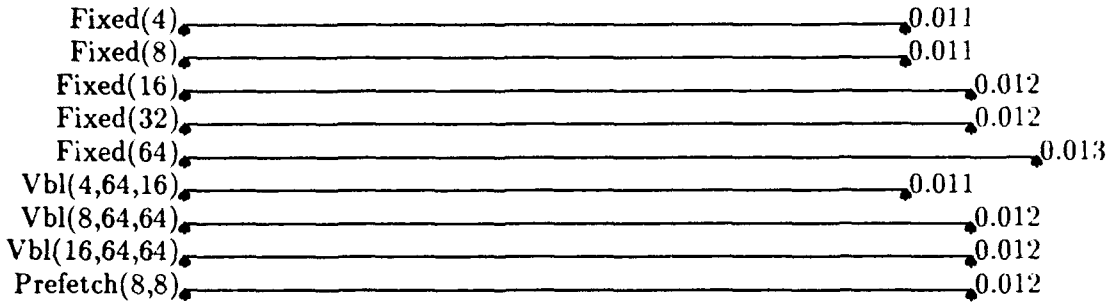


Figure A.11.2: Data Words Transferred Per Reference - *sorbyr*

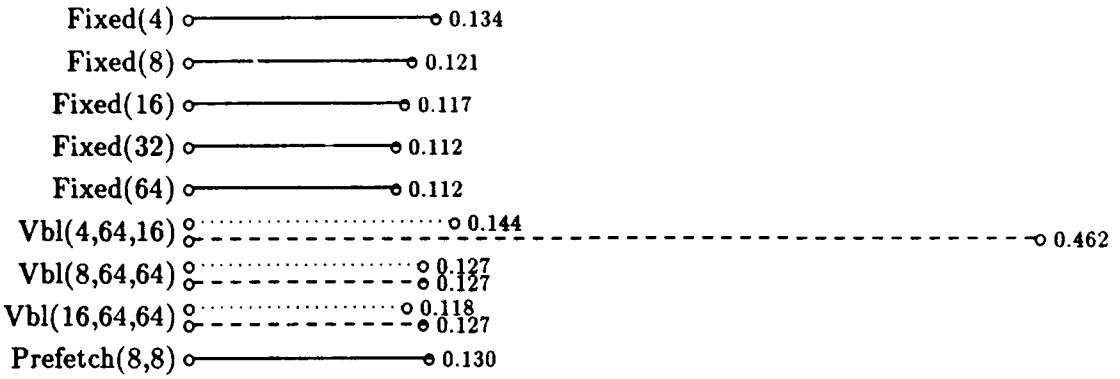


Figure A.11.3: Cache Size (bits per reference) - *sorbyr*. For adjustable caches dashed line gives the size of the *max-block* implementation, dotted line gives the size of the *min-block* implementation.

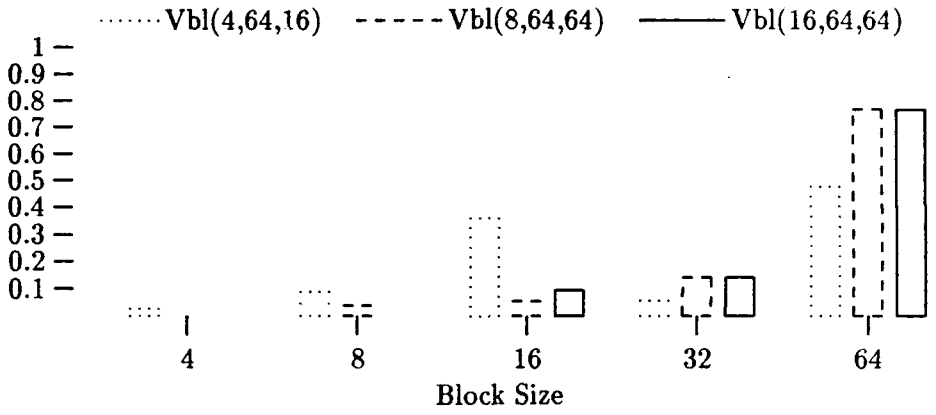


Figure A.11.4: Fraction of Block Transfers of Given Size - *sorbyr*

Table A.11: Split and Merge Statistics - *sorbyr*

Cache	Splits					Merges					Failed Merges
	Total	Block Size				Total	Block Size				
		8	16	32	64		4	8	16	32	
Vbl(4,64,16)	1066	3	1063			468		24	300	144	1989
Vbl(8,64,64)	738		8	18	712	12				12	1580
Vbl(16,64,64)	730			18	712	12				12	1571