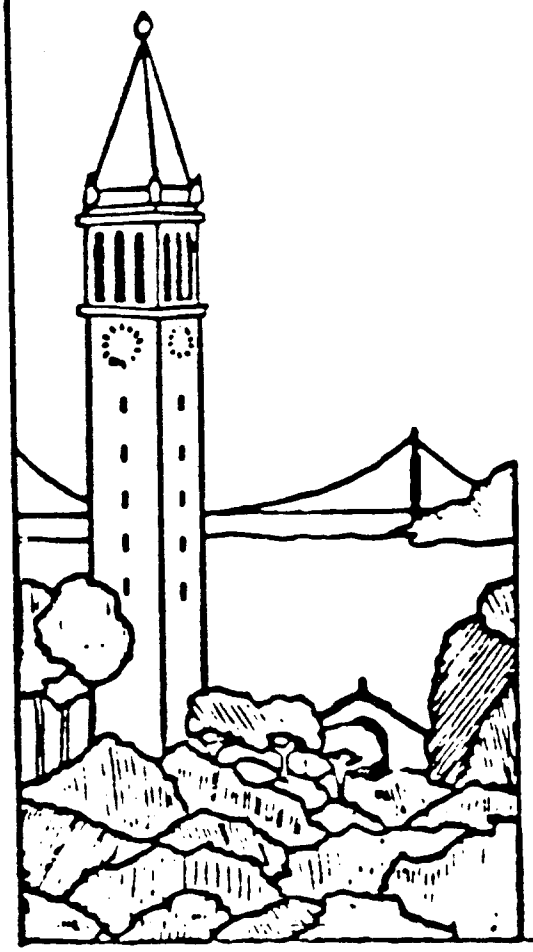
Aspects of Cache Memory and Instruction Buffer Performance

Mark Donald Hill



Report No. UCB/CSD 87/381

November 1987

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
NOV 1987		**00-00-1987 to 00-00-1987**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Aspects of Cache Memory and Instruction Buffer Performance	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Techniques are developed in this dissertation to efficiently evaluate direct-mapped and set-associative caches. These techniques are used to study associativity in CPU caches and examine instruction caches for single-chip RISC microprocessors. This research is motivated in general by the importance of cache memories to computer performance, and more specifically by work done to design the caches in SPUR, a multiprocessor workstation designed at U.C. Berkeley. The studies focus not only on abstract measures of performance such as miss ratios, but also include, when appropriate, detailed implementation factors, such as access times and gate delays. The simulation algorithms developed compute miss ratios for numerous alternative caches with one pass through an address trace, provided all caches have the same block size, and use demand fetching and LRU replacement. One algorithm (forest simulation) simulates direct-mapped caches by relying on inclusion, a property that all larger caches contain a superset of the data in smaller caches. The other algorithm (all associativity simulation) simulates a broader class of direct-mapped and set-associative caches than could previously be studied with a one-pass algorithm, although somewhat less efficiently than forest simulation, since inclusion does not hold. The analysis of set-associative caches yields two major results. First, constant factors are obtained which relate the miss ratios for set-associative caches to miss ratios for other set-associative caches. Then those results are combined with sample cache implementations to show that above certain cache sizes, direct-mapped caches have lower effective access times than set-associative caches, despite having higher miss ratios. Finally, instruction buffers and target instruction buffers are examined as organizations for instruction memory on single-chip microprocessors. The analysis focuses closely on implementation considerations, including the interaction between instruction fetches, instruction prefetches and data references, and uses the SPUR RISC design as the case study. Results show the effects of varying numerous design parameters, suggest some superior designs, and demonstrate that instruction buffers will be preferred to target instruction buffers in future RISC microprocessors implemented on single CMOS chips.

15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**183**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

ASPECTS OF CACHE MEMORY AND INSTRUCTION BUFFER PERFORMANCE

Mark Donald Hill

Computer Science Division
University of California
Berkeley, CA 94720

ABSTRACT

Techniques are developed in this dissertation to efficiently evaluate direct-mapped and set-associative caches. These techniques are used to study associativity in CPU caches and examine instruction caches for single-chip RISC microprocessors. This research is motivated in general by the importance of cache memories to computer performance, and more specifically by work done to design the caches in SPUR, a multiprocessor workstation designed at U.C. Berkeley. The studies focus not only on abstract measures of performance such as miss ratios, but also include, when appropriate, detailed implementation factors, such as access times and gate delays.

The simulation algorithms developed compute miss ratios for numerous alternative caches with one pass through an address trace, provided all caches have the same block size, and use demand fetching and LRU replacement. One algorithm (forest simulation) simulates direct-mapped caches by relying on inclusion, a property that all larger caches contain a superset of the data in smaller caches. The other algorithm (all associativity simulation) simulates a broader class of direct-mapped and set-associative caches than could previously be studied with a one-pass algorithm, although somewhat less efficiently than forest simulation, since inclusion does not hold.

The analysis of set-associative caches yields two major results. First, constant factors are obtained which relate the miss ratios for set-associative caches to miss ratios for other set-associative caches. Then those results are combined with sample cache implementations to show that above certain cache sizes, direct-mapped caches have lower effective access times than set-associative caches, despite having higher miss ratios.

Finally, instruction buffers and target instruction buffers are examined as organizations for instruction memory on single-chip microprocessors. The analysis focuses closely on implementation considerations, including the interaction between instruction fetches, instruction prefetches and data references, and uses the SPUR RISC design as the case study. Results show the effects of varying numerous design parameters, suggest some superior designs, and demonstrate that instruction buffers will be preferred to target instruction buffers in future RISC microprocessors implemented on single CMOS chips.

25 November 1987

ASPECTS OF CACHE MEMORY AND INSTRUCTION BUFFER PERFORMANCE

by

Mark Donald Hill

Acknowledgments

Many people deserve thanks for their help in making this dissertation possible. I want like to thank my committee, Alan Smith, David Patterson and Ronald Wolff, for their many insightful suggestions that enhanced the quality of my research and this presentation of it. In particular, I'd like to thank Alan Smith for his stern evaluation of my work that contributed considerably to making the final product better, and David Patterson for his many years of cogent advice and for providing an environment conducive to productive research.

Thanks also to: Sue Dentinger and Susan Eggers for critiquing drafts of my entire thesis, Jim Thompson and David Wood for improving drafts of Chapters 1 and 2, Valerie King for assisting me with the proofs in Chapter 2, Ken Lutz for assisting me in developing the cache hit implementations, my friends and colleagues on the SPUR project for numerous inspiring technical conversations, and all my friends and family for encouraging me these many years.

I want to acknowledge and thank those who provided trace data: Dick Sites at Digital Equipment Corp.; Anant Agarwal at Stanford; Alan Smith, George Taylor, and David Wood at Berkeley; Joe Hull and Rollie Schmidt at Synapse; John Lee and Bill Harding at Amdahl; and Robert Henry at Washington.

The material presented here is based on research supported in part by the Defense Advanced Research Projects Agency monitored by Naval Electronics Systems Command under Contract No. N00039-85-C-0269, the National Science Foundation under grants CCR-8202591 and MIP-8713274, by the State of California under the MICRO program and by IBM Corporation, Digital Equipment Corporation, Hewlett Packard Corporation, and Signetics Corporation.

Finally, a special thanks to *el amor de mi vida* and *Cafe Roma*.

Table of Contents

List of Figures

List of Tables

1 Introduction

This thesis consists of three self-contained chapters that examine cache evaluation techniques, the effects of associativity on cache performance, and tradeoffs in the design of on-the-CPU-chip instruction caches. Each of chapters 2, 3 and 4 is a self contained presentation; in this introductory chapter, I provide a general overview of the problems considered. Here I explain what caches are and why they work, discuss the motivation behind my studies, present the system that inspired the studies, and give an overview of the thesis.

1.1. Cache Memory

A *cache* is small, fast buffer in which a system tries to keep those parts of the contents of a larger, slower memory that will be used soon. The purpose of a cache is to improve system cost-performance by providing the capacity of the large slow memory with close to the access time of the small, fast cache. This is possible only if most memory references can be serviced rapidly by the cache without the intervention of the slower memory. In most cases, the cache is successful, because of *temporal* and *spatial locality*, two properties of most real reference streams [Denn70]. Temporal locality states that future references are likely to be made to the same locations as recent requests, while spatial locality states that future references are also likely to be made to locations near recent references. Caches take advantage of temporal locality by retaining recently referenced information, while they exploit spatial locality by loading and retaining (blocks of) information surrounding recent references.

A *CPU cache* is a cache of main memory [Smit82]. Like caches in general, CPU caches are faster and smaller than the memory they buffer. CPU caches are usually five to 20 times faster and 50 to 1000 times smaller than main memory. Because of their need for extremely high speed, CPU caches are managed entirely by hardware, and for this reason, CPU cache access and management policies must be relatively simple. Table 1-1 provides some succinct definitions for the basic CPU cache terminology used throughout this thesis. Other terms will be defined the first time they are used.

Term	Definition
reference	A request by the processor to read or write a memory location. (synonyms: request, access, processor reference, memory reference)
cache	A small, fast memory that holds active parts of a larger, slower memory. The capacity of a cache is the *cache size*.
hit, miss	References found in the cache are said to *hit;* those not found to *miss*.
memory	A larger, slower memory that provides data on cache misses.
block frame	A location in the cache that holds cached data, an associated address tag and state bits. The capacity of a block frame is the *block size*. (synonym: block)
block	Data from memory that fills a block frame. (synonyms: line, sector)
sub-block	A datum transferred from memory to the cache. The term is not used in the typical case when the transfer size is equal to the block size. (synonyms: subsector, block)
fetch algorithm	The method used to determine when and which datum should be brought into the cache. (synonym: prefetch policy)
demand fetch	A fetch algorithm that loads data only in response to cache misses. (synonym: no prefetching)
prefetching	Fetch algorithms that sometimes load data before it is referenced.
placement algorithm	The method used determine where a block may reside in a cache; often selects the *set* of a reference. (synonyms: placement policy, cache organization)
set	A collection of block frames in which a block can reside.
associativity	The number of block frames in each set. (synonyms: set size, degree of associativity)
n-way set-associative	A placement algorithm that divides a cache's block frames into more than one set of n block frames each (associativity n), where n is greater than one.
direct-mapped	A placement algorithm with single-block sets (associativity one). (synonym: one-way set-associative)
fully-associative	A placement algorithm with one set.
replacement algorithm	The method used to determine which block to replace when a new block is loaded. With set-associative placement, only blocks that reside in the set of the new block are considered for replacement. (synonym: replacement policy)
LRU	A commonly used replacement algorithm that replaces the *least recently used* block, that is, the one last referenced longest ago.

Table 1-1. Selected Basic CPU Cache Terminology.

CPU caches have been studied extensively (for a bibliography see [Smit86]), because properly designed caches have proven effective at increasing the performance or lowering the cost of many systems. CPU caches continue to be worth studying, because their importance to system cost-performance is increasing, and because technological improvements are altering the characteristics of well-designed caches. I expect the importance of CPU caches to increase in the coming decade as technological improvements widen the gap between CPU cycle times and main memory access times and as multiprocessors come into common use. An effect of the changing characteristics of CPU caches is that the foci of many early cache studies are no longer relevant.

The method I use to study CPU caches, *trace-driven simulation*, is commonly used to evaluate CPU caches. A trace-driven simulator is a program that models one or more caches in response to a *trace*, which is a dynamic series of processor references. The merits of trace-driven simulation are discussed in Chapter 3 and by Smith [Smit82, Smit85].

1.2. Computer Performance Architecture

The first goal in the design of any computer system is to make it operate correctly. A second goal is to achieve some level of performance at minimal cost. I concentrate on this second goal and refer to it as *computer performance architecture.*

A difficult problem for computer performance architects modeling a system is to select the appropriate level of abstraction. Models at too low a level of abstraction are tedious and time-consuming to build, and have limited generality. Building a complete register-transfer description of a multiprocessor before one has established that the interconnection network has adequate bandwidth, for example, suffers from these problems. Models at too high a level of abstraction can leave out details that are important in practice. RISC processors, for example, have proven to be effective despite register-transfer models that show that they must execute more instructions than do processors with more complex instruction sets [Patt85]. These models mispredict, because they assume that alternatives have the same cycle time and similar implementation costs, when in fact RISCs often have shorter cycle times and reduced implementation costs. Another problem computer performance architects face is that the appropriate level of abstraction does not remain constant; it begins high and then decreases as a system design develops.

The performance evaluation in this thesis is motivated by analysis done to design the caches in SPUR (a multiprocessor workstation described in the next section). In this thesis I use a level of abstraction higher than was used for the SPUR design studies, but lower than is used in many studies where all implementation considerations are ignored. I increase the level of abstraction so that these studies apply to similar caches in similar systems. I do not, however, raise the level of abstraction to the point where it can be said that my results apply to all caches. I believe the drawback of this reduced generality is more than compensated for by the increased realism and accuracy of the work.

1.3. SPUR

SPUR (Symbolic Processing Using RISCs) is a multiprocessor workstation being developed at U.C. Berkeley as a vehicle for conducting parallel processing research [Hill86]. A goal of the design effort was to build a straight-forward system, and then use that system as a framework for retrospectively studying less conservative system components. For this reason, part of this thesis re-examines the SPUR cache and instruction buffer (IB). Figures 1-1 and 1-2 show the SPUR system and a SPUR processor.

Each SPUR processor contains two caches. The first, the SPUR cache, is a 128K-byte board-level cache. It caches instructions and data in direct-mapped 32-byte blocks to reduce system bus traffic, memory contention, and effective memory access time. The SPUR cache is tagged with virtual addresses, rather than physical addresses, so that address translation is not necessary on cache hits. On cache misses, virtual addresses are translated into physical addresses before accessing shared memory.

The second cache in the SPUR processor, the SPUR instruction buffer (IB), is a 512-byte instruction cache on the CPU chip. It contains 16 direct-mapped blocks of eight instructions each (32 bytes). Valid bits are associated with each instruction word. On a miss, only a single instruction is loaded. Prefetching is used to load instructions into the rest of the block, in parallel with subsequent processor references.

1.4. Thesis Overview

Chapters 2, 3 and 4 are written as stand-alone discussions rather than as strongly integrated parts of a whole. For the reason, they can be read separately and in any order.

Chapter 2, *Simulation Techniques for Direct-Mapped and Set-Associative Caches,* discusses methods for the trace-driven simulation of numerous alternative single-level caches. New simulation algorithms are needed, because the established method, *stack simulation* [Matt70], is not efficient for simulating numerous caches of restricted associativity (e.g, the direct-mapped and two-way set-associative cache designs I studied for SPUR).

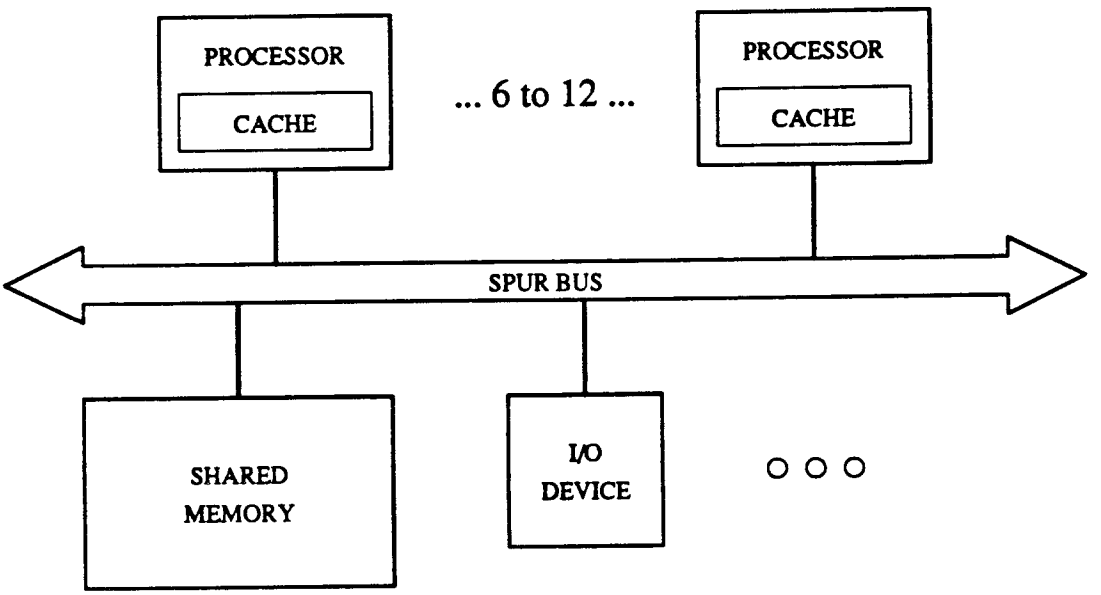


Figure 1-1. SPUR Workstation System.

SPUR (Symbolic Processing Using RISCs) is a workstation for conducting parallel processing research. SPUR contains 6 to 12 high-performance homogeneous processors connected with a shared bus. Each processor is a general-purpose RISC processor that provides some support for Common Lisp and IEEE floating-point. The number of processors is large enough to permit parallel processing experiments, but small enough to allow packaging as a personal workstation. Each of the custom processors contains a large cache to reduce the bandwidth required from the bus and shared memory. Standard NuBUS I/O devices and gateways to other busses are also attached to the SPUR Bus to complete the system. (Adapted from Figure 1 in [Hill86].)

Chapter 2 begins by examining *inclusion* and *set-hierarchy*, two properties of a collection of caches that facilitate the efficient simulation of alternative designs. Inclusion holds when all larger caches always contain a superset of the data in smaller caches [Matt70]. Mattson et al. show when inclusion holds for caches with the same block size, demand fetch and a fixed number of sets. I extend their results to show when inclusion holds for caches with a variable number of sets. Set-hierarchy, a property I introduce, holds if data that map to the same set in larger caches always maps to the same set in smaller caches. I find that inclusion and set-hierarchy hold for many series of direct-mapped caches, but that only set-hierarchy holds for most series of set-associative caches.

I develop an efficient algorithm for simulating direct-mapped caches, called *forest simulation*, that uses inclusion and set-hierarchy, and extend a previously published algorithm for set-associative caches, called *all-associativity simulation*, that does not use inclusion, but can use set-hierarchy. I implement stack, forest, and all-associativity simulation, and find that all-associativity simulation is fastest for evaluating numerous large direct-mapped and set-associative CPU caches, while forest simulation is fastest if only direct-mapped caches are being studied.

One impact of the work in Chapter 2 is that the efficiencies achieved enabled the scope of Chapter 3's studies (described below) to be expanded.

Chapter 3, *The Effect of Set-Associativity on Cache Memory Performance*, uses miss ratio and effective access time to evaluate caches of varying associativity. I examine associativity, because few studies have concentrated on it, and because technological advances are altering the balance between the benefits of higher associativity and the costs of implementing it.

I use the algorithms developed in Chapter 2 to compute the miss ratio of many caches with numerous traces. My results confirm the well-known facts that increasing associativity reduces miss ratio by amounts that diminish as associativity gets larger. I show further, by examining the *ratios* of miss ratios, that decreasing associativity causes relative changes in miss ratio that do not vary

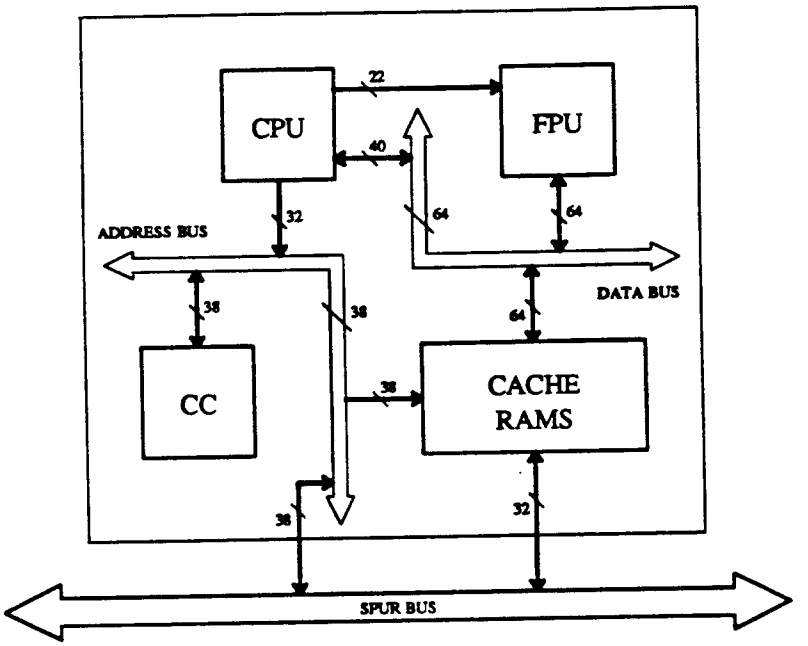


Figure 1-2. SPUR Processor.

A SPUR processor is implemented on a single board that contains three custom VLSI chips and 200 standard chips. The three custom chips are the CPU, the cache controller (CC) and the floating-point coprocessor (FPU). Off-the-shelf chips are used to hold the state, address tags, and data of the SPUR cache (CACHE RAMS), and to connect functional components together (not shown).

The CPU chip is a custom VLSI chip with a reduced instruction set. Like RISC II [Kate83], the SPUR CPU uses a simple and uniform pipeline, hard-wired control, and a large register file; it attempts to issue a new instruction every cycle. The SPUR CPU differs from RISC II because of the addition of a 512-byte instruction buffer, a fourth execution pipeline stage, a coprocessor interface, and support for Lisp tagged data.

The CC chip manages the SPUR cache, a 128K-byte, direct-mapped, mixed cache with 32-byte blocks. This includes handling cache accesses by the CPU, performing address translation, accessing shared memory over the SPUR Bus, and maintaining cache consistency.

The final custom chip is the floating-point coprocessor, which supports the full IEEE standard 754 for binary floating-point arithmetic without microcode control. Common operations are executed by the FPU under hard-wired control. Infrequent operations cause traps and are handled by software. (Adapted from Figure 2 in [Hill86].)

significantly as caches get larger, but that do change with cache type and block size. I also use these ratios of miss ratios to extend Smith's design target miss ratios to caches of varying associativity [Smit85, Smit87].

Miss ratio comparisons of cache of varying associativity have generally ignored implementation concerns. For small caches (less than 32K bytes), ignoring implementation details is reasonable, since the miss ratio change cause by varying associativity is large. As cache size increases, however, absolute miss ratios become small, and therefore the difference in miss ratios between designs becomes less important than their access time differences.

I use *effective access time* to take into account the effects of varying implementation delays. Effective access time is the average latency, as seen by the processor, on its references. I show that a direct-mapped cache with a higher miss ratio but a smaller access time than a set-associative cache can have a superior effective access time. Next by incorporating the above miss ratio results with three implementation examples, I show that some practical direct-mapped caches of 32K bytes and larger have lower effective access times than comparable set-associative caches.

The impact of the work in Chapter 3 is that for large caches, as was found for RISCs, the implementation advantages of simplity can overwhelm the architectural advantages of more complex designs. This work also illustrates the benefit of lower levels of abstraction; had I ignored implementation details and assumed that all caches have the same access time, I would have reached an incorrect conclusion.

Chapter 4, *Instruction Memory on a Single-Chip RISC*, examines the design of instruction buffers and target instruction buffers to reduce the effective instruction access time of a single-chip implementation of RISCs. I evaluate instruction memories with a trace-driven simulator that includes a detailed model of a RISC pipeline and the potential interference between instruction fetches, instruction prefetches and data references.

An *instruction buffer* is a cache for instructions only. Among other things I find that the performance of an instruction buffer is strongly affected by its size and ability to rapidly fetch and prefetch instructions from off of the CPU chip. I propose improvements to the SPUR instruction buffer, applicable to instruction buffers on other RISC CPU chips, which halve its miss ratio and reduce effective access time by 20 percent.

A *target instruction buffer* holds one or more instructions at recent branch targets, but does not try to cache all recently executed instructions (e.g., the AMD 29000 Branch Target Cache [Adva87]). As with instruction buffers, I find that target instruction buffer performance improves when instructions can be fetched and prefetched more rapidly from off-chip. In contrast to instruction buffers, I discover that target instruction buffer performance does not improve rapidly as buffer size is increased.

Finally, I compare instruction buffers and target instruction buffers in CPUs connected to single- and multiple-cycle off-chip memories. With single-cycle off-chip memories, I show that target instruction buffers give comparable or better performance, unless on-chip memory size is large ($\geq$ 8K bytes). As off-chip memories become relatively slower, the performance of instruction buffers improves relative to that of target instruction buffers, because instruction buffers rely more on caching and less on prefetching than do target instruction buffers.

The impact of the work in Chapter 4 is: (1) it provides detailed assistance to the designers of instruction memory on RISC CPU chips, and (2) it demonstrates that instruction buffers will be preferred to target instruction buffers in future CMOS microprocessors, where available buffer size will be large and single-cycle off-chip accesses unlikely. Nevertheless, target instruction buffers may be appropriate for other technologies, such as Gallium Arsenide.

1.5. References

[Adva87] Advanced Micro Devices, Am29000 User's Manual (1987).

[Denn70] P. J. Denning, Virtual Memory, *Computing Surveys*, 2, 3 (September 1970).

[Hill86] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, Design Decisions in SPUR, *IEEE Computer*, 19, 11 (November 1986).

[Kate83] M. G. H. Katevenis, R. W. Sherburne, D. A. Patterson and C. H. Séquin, The RISC II Micro-Architecture, *Proc. VLSI 83 Conference*, Trondheim, Norway (August 1983).

[Matt70] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger, Evaluation techniques for storage hierarchies, *IBM Systems Journal*, 9, 2 (1970), 78 - 117.

[Patt85] D. A. Patterson, Reduced Instruction Set Computers, *Comm. ACM* (January 1985), 8-21.

[Smit82] A. J. Smith, Cache Memories, *Computing Surveys*, 14, 3 (September, 1982), 473 - 530.

[Smit85] A. J. Smith, Cache Evaluation and the Impact of Workload Choice, *Proc. Twelfth International Symposium on Computer Architecture* (June 1985).

[Smit86] A. J. Smith, Bibliography and Readings on CPU Cache Memories and Related Topics, *Computer Architecture News* (January 1986), 22-42.

[Smit87] A. J. Smith, Line (Block) Size Choice for CPU Caches, *IEEE Trans. on Computers*, C-36, 9 (September 1987).

2 Simulation Techniques for Direct-Mapped and Set-Associative Caches

2.1. Introduction

Design and research questions regarding memory hierarchies are often investigated with trace-driven simulation of several design alternatives. For this reason, Mattson et al. developed the *stack simulation* technique for simulating many caches with one pass through an address trace [Matt70]†. Stack simulation can evaluate alternative caches of many sizes if all have the same number of sets, the same block size, do no prefetching, and use a *stack* replacement algorithm (e.g., LRU and RANDOM). Caches in a single stack simulation all have different associativities, however, since associativity is cache size in blocks (which varys in a stack simulation) divided by the number of sets (which is fixed). Design and research questions regarding CPU caches often examine caches of differing sizes, but fixed associativity [Smit78], [Clar83], [Good83], [Haik84], [Hill84] and [Puza85]. Consequently the evaluation of alternative CPU cache designs can require numerous stack simulations.

To reduce the number of simulations required, I have developed efficient one-pass trace-driven simulation algorithms for evaluating caches having differing numbers of sets. In some cases I reduce simulation time by using *inclusion* [Matt70]. I say cache C_2 *includes* cache C_1 if cache C_2 contains a superset of the blocks in cache C_1 after any series of references. A simulation of alternative cache designs can take advantage of inclusion by searching for a reference in cache C_1 first, cache C_2 second, and then in other caches that include cache C_2. When a reference is found, a hit can be recorded for that

† Mattson et al. describe stack simulation with the terminology of virtual memory, e.g., pages and memory sizes. I use the terminology of CPU caches -- blocks (lines) and cache sizes -- because direct-mapped and set-associative placement is typically used only for CPU caches. I use *cache* to refer to any memory used to buffer the active blocks of a slower memory, and use *CPU cache* to refer specifically to a cache that buffers blocks from main memory.

cache and (implicitly) for all caches that include that cache. Mattson et al. show when inclusion holds for caches with the same number of sets, and use inclusion to make stack simulation efficient.

Here I show when inclusion holds for caches having differing number of sets. I find inclusion holds between practical direct-mapped (one-way set-associative) CPU caches, but that it does not hold in general between practical set-associative CPU caches. Since direct-mapped caches are important (see Chapter 3 and [Bell74, Patt83, Stre76]), I develop an algorithm, called *forest simulation*, for simulating alternative direct-mapped caches that takes advantage of inclusion. Since set-associative caches are also important, I describe an algorithm, called *all-associativity simulation*, for simulating alternative set-associative caches that does not take advantage of inclusion. I allow alternative caches to use arbitrary functions to map references to sets. I also show that faster simulation times can be achieved when the functions that map references to sets obey a property called *set hierarchy* (described in Section 2.3). My algorithm is a generalization of an algorithm for simulating set-associative caches that map references to sets with *bit selection* [Matt70, Trai71]. A cache that uses *bit selection* contains a power of two number of sets and selects the set of a reference with the least-significant bits of the reference's block number.

The rest of this paper is organized as follows: section 2.2 discusses background; section 2.3 proves results about inclusion for set-associative caches; sections 2.4 and 2.5 develop forest and all-associativity simulation algorithms; and sections 2.6 and 2.7 contain experimental results and conclusions.

2.2. Background

This section reviews set-associative caches, introduces stack algorithms, describes and analyzes linked-list stack simulation and describes more efficient methods of stack simulation.

2.2.1. Set Associative Caches

A fully-associative cache allows any block to reside in any block frame. An n-way set-associative cache of c blocks uses a set-mapping function f to partition all blocks in main memory into a number of equivalence classes, and allows at most n blocks from each equivalence class to be simultaneously resident. The block frames that hold blocks from one equivalence class are called a *set*. The number of block frames in a set, n, is called the *associativity* (or *degree of associativity* or *set size*). The number of equivalence classes in the image of f, called the *number of sets*, is always equal to c/n, the number of blocks in a cache divided by its associativity. The advantage of a set-associative cache with respect to a fully-associative cache of the same size is that n block frames rather than c block frames must be searched on each reference. The disadvantage of a set-associative cache is that it restricts which blocks can be simultaneously resident. For example, an n-way set-associative cache cannot contain the $n+1$ most-recently-referenced blocks that map to one set. Figure 2-1 illustrates set-associative mapping and discusses my notation for caches.

The most-commonly used set-mapping function is *bit selection*, because it can be implemented with no logic or delay. In bit selection, several low order bits of the block number are used to select the set. Bit selection requires that the number of sets be a power of two. For example, the set of block x is a cache with 2^i sets that uses bit selection is $f(x) = x \text{ rem } 2^i$, where $x \text{ rem } 2^i$ is the remainder of dividing x by 2^i.

2.2.2. Stack Algorithms

The seminal paper on memory hierarchy simulation is Mattson et al. [Matt70]. It introduces *stack simulation* as an efficient technique for evaluating a series of fully-associative caches and obey the inclusion property. Since a set-mapping function partitions blocks into equivalence classes and set-associative caches do not allow blocks from different classes to interact, each set of a set-associative cache operates as an independent fully-associative cache. For this reason stack simulation can be applied to a set-associative caches that use the same set-mapping function.

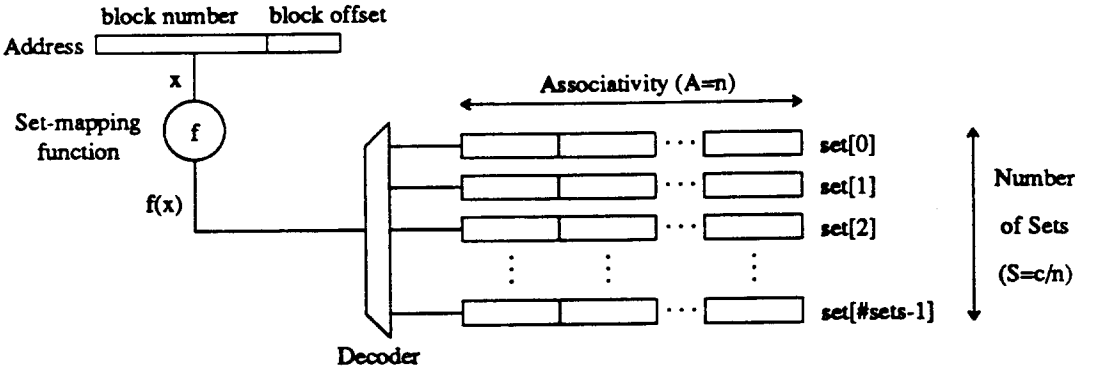


Figure 2-1. Set-Associative Mapping.

This figure illustrates set-associative mapping in an n-way set-associative cache of c blocks with set-mapping function f. If a block x is present, it is in one of the n block frames in set $f(x)$ (one row). The number of elements in a single set is the associativity (degree of associativity, set size, A). The number of values in the image of f (number of rows) is the number of sets in the cache ($S=c/n$). The associativity times the number of sets is always equal to the cache's size in blocks. A cache is direct-mapped if $A=1$; it is fully-associative if $S=1$.

I denote the above cache with "C(A=n, S=c/n, F=f)," where A, S and F are cache parameters "associativity," "number of sets" and "set-mapping function." When comparing caches I omit listing parameters that do not vary. For example, I use "C(A=1)" and "C(A=2)" to contrast a direct-mapped and a two-way set-associative cache that are otherwise similar. When differences are clear, I use subscripts for distinguishing caches (e.g., "C_1" and "C_2"). Finally I use "c_i" (lower case) to represent the number of blocks in cache "C_i" (upper case). Thus "C_i" represents all attributes of cache C_i, while "c_i" represents only the number of blocks in cache C_i.

Stack simulation is efficient because it takes advantage of *inclusion*, which is the property that, after any series of references, each larger cache simulated contains a superset of the blocks resident in all smaller caches. Inclusion may seem trivially true, but it is not. For example, a series of caches managed with FIFO (first in first out) replacement do not always obey inclusion. Consider a series of references to blocks 1, 2, 3, 1, and 4. At the end of this sequence, a two-block cache will contain blocks 1 and 4 while a three-block cache will contain 2, 3, and 4, but not block 1.

Assuming no prefetching and fixed block size, Mattson et al. show that inclusion holds between caches using the same set-mapping function for a class of replacement algorithms called *stack algorithms*. LRU, RANDOM and OPTIMUM [Bela66] are the principal, interesting stack algorithms.

A stack simulation of caches $C(A=k, F=f)$ for $k = 1$ to n uses a stack of n nodes for each set in the image of f, and an array of n *distance* counters. If I assume LRU replacement, each stack conceptually lists the most-recently-referenced n blocks for its set. Stacks in simulations of other stack replacement algorithms list blocks in order of descending priority, where priorities are defined so that blocks with a lower priority are preferred for replacement with respect to blocks with a higher priority. Each counter *distance[k]* contains the number reference so far to the k-th most-recently-referenced block. For each reference x, stack simulation performs three steps: *FIND*, *UPDATE* and *METRIC*.

FIND — Locate block x in stack $f(x)$. I say a reference is found at *distance* k if it is the k-th element in the stack, and at distance infinity (∞) if it not found.

METRIC — Increment counters *distance[k]* and N, where N is the number of references. At the end of simulation, the miss ratio of cache $C(A=k, F=f)$ is $1 - \sum_{j=1}^{k} distance[k]/N$. Metrics can be also be maintained by keeping counters only for specific cache sizes of interest. This will save space, but increase the time required to determine what counter(s) to increment.

UPDATE — Update the stack to reflect the contents of all caches after the reference to x. See Mattson et al. for what is required with an arbitrary stack algorithm. For LRU, x must be moved

from it old position (if any) to the top of stack $f(x)$, all blocks x passes must be moved down one position, and all other blocks must not move. If x was not previously referenced, x must be inserted at the top of stack $f(x)$, and all other blocks in stack $f(x)$ must be moved down one position.

2.2.3. Linked-List Stack Simulation

Here I describe stack simulation with the stack for each set implemented with a linked-list. This is commonly done for CPU cache simulations, because it is simple to implement and has adequate performance since the referenced block is usually found in the first few elements of the stack. I assume LRU replacement, because it is commonly used; the arguments that follow can also be extended to other stack replacement algorithms. Figure 2-2 shows an example eight-entry stack before and after a reference.

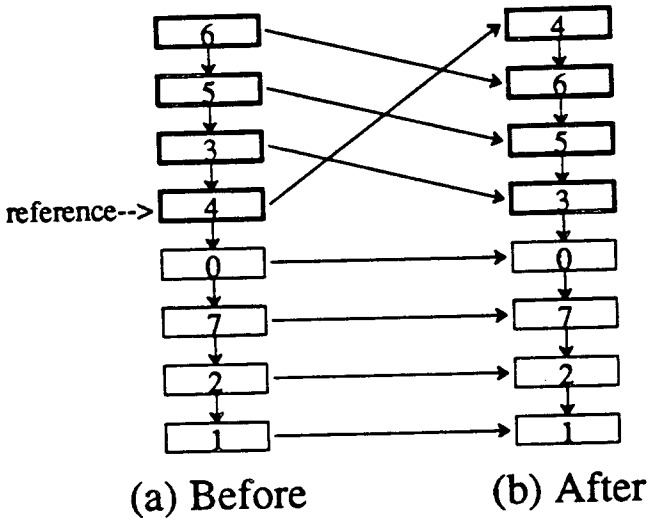


Figure 2-2. Stack Simulation Example.

The left stack (a) shows an LRU stack for one set after a series of references to that set. Information in the stack reveals that block 6 is in this set of a direct-mapped cache (one block per set); blocks 6 and 5 are in a two-way set-associative cache; blocks 6, 5, and 3 are in a three-way cache; ...; and blocks 0 through 7 are in an eight-way cache.

Let the next reference that maps to this set be to block 4. The blocks in bold are examined to find block 4. The search stops when block 4 is found or the stack is exhausted. Since block 4 is located (coincidentally) at stack depth 4, a miss is recorded for all caches smaller than four blocks, and a hit is recorded for all caches 4 blocks or larger. The right stack (b) shows the stack after it has been updated with LRU replacement; the blocks in bold have moved.

The pseudo-code in Figures 2-3 and 2-4 illustrate the storage and the per-reference processing required by linked-list stack simulation. The implementation of FIND (not shown) merely walks down the link-list $f(x)$ until reference x is found or the linked-list is exhausted. If x is found, the implementation of UPDATE (also not shown) changes two pointers to move x to the head of the linked-list $f(x)$. Otherwise, it allocates a new node for x, either from a free list or by reclaiming the last block in the list, and puts the node at the head of the linked-list.

The analysis of the time to simulate each reference is some constant, $O(1)$, that includes the time to read the reference, plus the number of iterations within FIND. Let δ_k be the probability that a reference is found at stack depth k, let δ_∞ be the probability that the reference is not found, i.e., this is the first reference to that block, and let N be the number of references in the trace. FIND uses k iterations to find a reference at stack distance k, and $O(N\delta_\infty)$ iterations for stack distance ∞ where $O(N\delta_\infty)$ is the size of the entire stack. In practice, the average stack size is much smaller than the number of unique blocks in the trace, $N\delta_\infty$, because the unique blocks are distributed across a 100 of more sets. The time

```
integer max_assoc -- maximum stack size
function f(x) -- a set-mapping function
integer number_of_stacks -- number of sets induced by f(x)
integer N -- number of references
-- distance counts so that m(C(F=f, A=k) = 1 - Σ_{j=1}^{k} distance[j]/N
integer distance[1:max_assoc]
define stacknode_type {
        integer block_number
        stacknode_type *next
}
stacknode_type *stack[0:number_of_stacks-1] -- top of stack pointers
-- pool of dynamically linked stacknodes
stacknode_type stacknodes[1:number_of_stacks*max_assoc]
```

Figure 2-3. Stack Simulation Storage.

```
For each reference x {
        read(var x)
        N++
        stack_number = f(x)
        -- Walk down stack until x is found or stack is exhausted.
        -- If found, return stack distance and pointers to stacknode containing x.
        -- Otherwise set stack distance to max_assoc+1 and point to LRU stacknode.
        found = FIND(x, stack_number, var stack_distance,
                var previous_node_pointer, var node_pointer)
        -- METRIC
        if (found) distance[stack_distance]++
        -- If was found, move the stack node of x to the top of its stack.
        -- Otherwise, store x in LRU stacknode and move it to the top of its stack.
        UPDATE(x, stack_number, found, previous_node_pointer, node_pointer)
}
```

Figure 2-4. Stack Simulation.

to process a reference is of order:

$$\sum_{k=1}^{\infty} k\,\delta_k + O(N\delta_\infty)\delta_\infty + O(1). \qquad \text{Eq. 2-1.}$$

The first term, called the *mean stack distance*, is the average number of distinct blocks since the last reference to the referenced block. If one is simulating only caches with associativity $\leq kmax$, then no stack node need to be retained beyond distance $kmax$. This reduces the simulation time to:

$$\sum_{k=1}^{kmax} k\,\delta_k + kmax^* \sum_{k=kmax+1}^{\infty} \delta_k + kmax^*\,\delta_\infty + O(1). \qquad \text{Eq. 2-2.}$$

Bounding stack size can significantly reduce simulation time of set-associative CPU caches, where $kmax$ rarely exceeds eight. However, for fully-associative caches, $kmax$ is equal to the number of

blocks in the largest cache simulated. The run-time of linked-list stack simulation of fully-associative caches will be poor if either $\sum_{k=kmax+1}^{\infty} \delta_k$ or δ_∞ is large.

An analysis of the exact storage required for bounded linked-list stack simulation of even large CPU caches is uninteresting, because the storage required is small relative to modern main memories. For example, the storage required by the linked-list stack simulation pseudo-code in Figure 2-3 for simulating a direct-mapped 128K-byte cache, a two-way set-associative 256K-byte cache, a four-way 512K cache and a eight-way 1M cache with 32-byte blocks is approximately equal to the number of blocks in the 1M-byte cache (32K) times 8 bytes per block, and is less than 300K bytes.

2.2.4. Other Stack Simulation Implementations

Bennett and Kruskal [Benn75] examine the paging behavior of a large data base. They find mean stack distances of 12 to 328 entries for varying page sizes. Bennett and Kruskal propose an algorithm for stack simulation using an m-ary tree and a hash table where the run-time per reference is approximately logarithmic in the number of blocks since the last reference to the current block. In contrast, the time per reference for linked-list stack simulation is linear in the number of distinct blocks since the last reference of the current block. Bennett and Kruskal conclude that their algorithm is of order ten times faster than linked-list stack simulation for mean stack distances of 150 entries. The storage requirements of the algorithm are large, but this is not important since the memory required is small relative to modern main memory sizes. The tree size is linear in the length of the address trace, N, and the hash table must be larger than the number of distinct blocks ($N\delta_\infty$). A simulation of 10 million references with 200,000 unique blocks requires only 3M bytes of storage if it uses two bits per reference and two words per unique block. Olken [Olke81] changes Bennett and Kruskal's algorithm by replacing their m-ary tree with an AVL tree (see [Thom87]).

Bennett and Kruskal's algorithm and Olken's algorithm use a hash table to learn about a block's history. A hash table can also be used in linked-list stack simulation to see if a block has ever been referenced. This reduces the time to process a previously unreferenced block from *kmax* to a constant, reducing simulation time to:

$$\sum_{k=1}^{kmax} k\,\delta_k + kmax^* \sum_{k=kmax+1}^{\infty} \delta_k + O(\delta_\infty) + O(1).$$

This change will significantly improve performance only if both *kmax* and δ_∞ are large, that is, both the degree of associativity and the fraction of the references to previously unreferenced blocks are large.

Thompson et al. [Thom86] examine each of these algorithms, and conclude that linked-list stack simulation performs best for most CPU cache simulations. Consequently, I will compare the performance of forest and all-associativity simulation with linked-list stack simulation only, and use *stack simulation* to refer to linked-list stack simulation.

2.3. Inclusion in Set-Associative Caches

Here I prove several theorems about inclusion for set-associative caches using (possibly) differing set-mapping functions. Recall that Mattson et al. [Matt70] discuss inclusion only in caches that use the same set-mapping function, and hence have the same number of sets (e.g., all are fully-associative). In this section, as in the rest of this chapter, I assume that all caches have the same block size, do no prefetching, and use LRU replacement. I want to use inclusion to rapidly simulate alternative single-level cache designs. Consequently when I discuss a large and a small cache, I am considering using one or the other in a memory system, not using both as components in a cache hierarchy.

Consider two caches, $C_1(A=n_1, F=f_1)$ and $C_2(A=n_2, F=f_2)$, with c_i blocks, associativities of n_i, and set-mapping functions f_i, for $i=1,2$. An important condition necessary for cache C_2 to *include* (the blocks of) cache C_1 is that all blocks mapping to the same set in C_2 map to the same set in C_1. That is, for all blocks x and y:

$$f_2(x)=f_2(y) \text{ implies } f_1(x)=f_1(y).$$

I call this condition *set hierarchy*, because it means that f_2 induces a finer partition on all blocks than does f_1. Assume also that each set-mapping function maps a large number of blocks ($\geq 2*\max(n_1, n_2)$) to each set. Set-mapping functions used in real caches, including bit selection, trivially meet this restriction.

For cache C_2 to include cache C_1, C_2 must be at least as large as C_1, otherwise inclusion will be violated as soon as C_1 is full. For cache C_2 to include a *different* cache C_1, C_2 must be strictly larger than C_1. I consider two caches to be equivalent if they always contain the same blocks, i.e., are identical up to placement of sets. Suppose cache C_1 and cache C_2 are the same size. For cache C_2 to include cache C_1, it must always contain a superset of cache C_1's blocks. Since cache C_2 contains the same number of blocks as C_1, it must always contain exactly the same blocks, and therefore is not a different cache†. For this reason I sometimes refer to cache C_2 as the "larger" cache.

Theorem 1

Cache $C_2(A=n_2, F=f_2)$ *includes* cache $C_1(A=n_1, F=f_1)$ if and only if $f_2(x)=f_2(y)$ implies $f_1(x)=f_1(y)$ (set hierarchy) and $n_2 \geq n_1$ (non-decreasing associativity).

Proof

==>. Suppose cache C_2 includes cache C_1 and $f_2(x_1)=f_2(x_2)=\cdots=f_2(x_{2n_2})$ for some $2n_2$ blocks x_1, ..., x_{2n_2}. The x_j's exist, because I assume each set-mapping function maps a large number of blocks to each set. To demonstrate that both set hierarchy and non-decreasing associativity are necessary for inclusion, I show that one of the x_j's must be in cache C_1 but not in larger cache C_2 if either (1) set hierarchy is false or (2) set hierarchy holds, but the larger cache has the smaller associativity.

(1) With set hierarchy false, let the $2n_2$ x_j's be chosen so that at least one block, y, maps to a different set in cache C_1 than does x_1 (i.e., $f_1(y)\neq f_1(x_1)$). Either (a) less than n_2 of the x_j's map to $f_1(x_1)$ or (b) n_2 or more of the x_j's map to $f_1(x_1)$. For (a), reference x_1 and the $\geq n_2$ blocks that *do not* map to $f_1(x_1)$. Inclusion is now violated since x_1 is in cache C_1, but not in larger cache C_2. It is in cache C_1 since all other blocks referenced map to other sets; it is replaced in n_2-way set-associative cache C_2, since at least n_2 other blocks mapping to its set are more-recently-referenced. For (b), reference y and the $\geq n_2$ blocks that *do* map to $f_2(x_1)$. Inclusion is now violated since y is in cache C_1, but not in the larger cache C_2.

(2) Since set hierarchy holds and $f_2(x_1)=f_2(x_2)=\cdots=f_2(x_{n_2+1})$, I know that $f_1(x_1)=f_1(x_2)=\cdots=f_1(x_{n_2+1})$. Reference x_1 through x_{n_2+1} in succession. Inclusion is now violated since x_1 is in n_1-way set-associative cache C_1 ($n_1>n_2$ implies $n_1\geq n_2+1$), but not in n_2-way set-associative cache C_2.

<==. Suppose set hierarchy and $n_2 \geq n_1$. Initially both caches are empty and inclusion holds, because everything (nothing) in cache C_1 is also in cache C_2. Consider the first time inclusion is violated, i.e., some block is in cache C_1 that is not in cache C_2. This can only occur when some some block y is replaced from cache C_2, but not from cache C_1. A block y can only be replaced from cache C_2 if n_2 blocks, x_1 through x_{n_2}, all mapping to $f_2(y)$, are referenced after it. By set hierarchy, $f(y)=f(x_1)=\cdots=f(x_{n_2})$. Since $n_2 \geq n_1$, y must also be replaced in cache C_1.

QED.

Theorem 1 states that inclusion holds between two set-associative caches only if the two caches obey set hierarchy and non-decreasing associativity. In Section 2.5 I show that set hierarchy and non-decreasing associativity are too restrictive to permit inclusion to hold between many pairs of set-associative caches, and then I describe an algorithm for simulating numerous set-associative caches does not try to take advantage of inclusion.

---

† This only shows that the caches contain the same blocks when they are full. Since I assume no prefetching, it is easy, but uninteresting, to show that they also contain the same blocks when filling.

I next show that the *includes* relation is a partial ordering of the set of set-associative caches (with the same block size, that do no prefetching, and use LRU replacement). A partial ordering differs from a total ordering (e.g., "≤" on the set of real numbers), because some elements may not be comparable (i.e., neither C_2 includes C_1 nor C_1 includes C_2). While establishing *includes* as a partial ordering is mostly of theoretical interest, it does enable transitivity to be used in the proof of Theorem 3.

Theorem 2

The *includes* relation is a partial ordering of the set of caches.

Proof

I must show that *includes* is reflexive (C_1 includes C_1), antisymmetric ((C_2 includes C_1 and C_1 includes C_2) implies $C_2 = C_1$) and transitive ((C_3 includes C_2 and C_2 includes C_1) implies C_3 includes C_1).

Reflexive. A cache includes another if it contains a superset of the blocks of the other. Clearly C_1 includes C_1, since two identical caches always contain the same blocks.

Antisymmetric. Suppose C_2 includes C_1 and C_1 includes C_2. Therefore cache C_2 must always contain a superset of the blocks in cache C_1, and cache C_1 must always contain a superset of the blocks in cache C_2. Since *superset* is antisymmetric, both caches must always contain the same blocks, and therefore are equivalent.

Transitive. Suppose $C_3(A=n_3, F=f_3)$ includes $C_2(A=n_2, F=f_2)$ and $C_2(A=n_2, F=f_2)$ includes $C_1(A=n_1, F=f_1)$. By Theorem 1, $n_3 \geq n_2$, $n_2 \geq n_1$, $f_3(x)=f_3(y)$ implies $f_2(x)=f_2(y)$, and $f_2(x)=f_2(y)$ implies $f_1(x)=f_1(y)$, for all blocks x and y. Since both relations "≥" and *implies* are transitive, $n_3 \geq n_1$ and $f_3(x)=f_3(y)$ implies $f_1(x)=f_1(y)$. By Theorem 1, C_3 includes C_1.

QED.

Next I consider caches using set-mapping functions of the form "$h(x)$ rem s," where $h(x)$ is a hash function whose image if the set of all block numbers, "rem" is the remainder operator, and s is the number of sets in a cache. I show that set hierarchy holds between two such caches if and only if the number of sets in the larger cache is a multiple of the number of sets in the smaller cache.

Theorem 3

Set hierarchy holds, that is, $f_2(x)=f_2(y)$ implies $f_1(x)=f_1(y)$, for set-mapping functions of the form $h(x)$ rem s_i if and only if s_1 divides s_2.

Proof

==>. Suppose set hierarchy holds, that is, $h(x) \text{ rem } s_2 = h(y) \text{ rem } s_2$ implies $h(x) \text{ rem } s_1 = h(y) \text{ rem } s_1$. Suppose s_1 does not divide s_2, then $s_2 \text{ rem } s_1 = k$ where $k \neq 0$. Let $h(x) = s_1 s_2$ and $h(y) = s_2(s_1 + 1)$. I know that there exist some block numbers x and y for which the above is true, because I require the image of hash function h be the set of all block numbers. For these values of $h(x)$ and $h(y)$, $h(x) \text{ rem } s_2 = h(y) \text{ rem } s_2 = 0$, but $h(x) \text{ rem } s_1 = 0$ and $h(y) \text{ rem } s_1 = s_2 \text{ rem } s_1 = k$ where $k \neq 0$. Thus, $h(x) \text{ rem } s_2 = h(y) \text{ rem } s_2$ is true while $h(x) \text{ rem } s_1 = h(y) \text{ rem } s_1$ is not. A contradiction. Therefore, s_1 must divide s_2 for set hierarchy to hold.

<==. Suppose s_1 divides s_2. By definition of divides, $s_2 = ns_1$ for some integer n. If $h(x) \text{ rem } s_2 = h(y) \text{ rem } s_2$, then $h(x) = x's_2 + k$ and $h(y) = y's_2 + k$ for some integers x', y' and k. Substitution yields $h(x) = x'ns_1 + k$ and $h(y) = y'ns_1 + k$. By definition of remainder, $h(x) \text{ rem } s_1 = h(y) \text{ rem } s_1 = k$. Thus $h(x) \text{ rem } s_2 = h(y) \text{ rem } s_2$ implies $h(x) \text{ rem } s_1 = h(y) \text{ rem } s_1$, or set hierarchy holds.

QED.

Theorem 3 allows us to prove that inclusion holds for many practical direct-mapped caches, including those using bit selection. Consider a series of direct-mapped caches, C_i, where each cache uses set-mapping function $f_i(x) = h(x) \text{ rem } c_i$ and each c_i divides c_{i+1}. By Theorem 3, set hierarchy holds between each pair of caches. Since set hierarchy holds and all associativities are equal (to one),

inclusion holds between each pair of caches by Theorem 1. Since inclusion is a partial ordering (Theorem 2), inclusion holds between all caches in the series. The above applies to series of direct-mapped cache that use bit selection, because for such caches $h(x)=x$ and each c_i divides c_{i+1} because both are powers of two. Consequently inclusion holds between direct-mapped caches that use bit selection.

Since inclusion holds for many direct-mapped caches and inclusion can be used to make simulations run more rapidly, I develop an algorithm for simulating direct-mapped caches that obey inclusion, which I present in the next section.

2.4. Simulating Direct-Mapped Caches with Inclusion

This section introduces *forest simulation* for evaluating direct-mapped caches that have the same block size and obey inclusion. Like stack simulation, forest simulation takes advantage of inclusion by searching for a block from the smallest to largest cache. When a block is found, a hit can be implicitly recorded in all larger caches. Forest simulation is so named because it uses a forest (a set of disjoint trees) rather than a stack to store cache blocks.

Let the direct-mapped caches be named $C_1, C_2, ..., C_L$. Assume that each cache C_i has c_i block frames and uses set-mapping function rem c_i. While forest simulation works for arbitrary set-mapping functions that obey set hierarchy, it is easier to describe it with set-mapping functions of the form rem c_i. Let $1 \le c_1 < c_2 < \cdots < c_L$ and c_i divide c_{i+1} for $i=1, L-1$. By the argument presented after Theorem 3, inclusion holds for these caches.

The key data structure in forest simulation is a forest of L levels. The number of trees in the forest is equal to the number of blocks in the smallest cache, c_1. The c_i nodes of level i represent the blocks in cache C_i. The branching factor between two levels is equal to the cache size of the larger level, divided by the cache size of the smaller level, c_{i+1}/c_i. The leaves represent the blocks in the largest cache, c_L. This forest can be implemented a heap containing twice as many nodes as there are blocks in the largest cache, since $c_{i+1}/c_i \le 2$ for all i implies $\sum_{i=1}^{L} c_i$ is less than $2*c_L$. For example, the heap location of block x a cache of c blocks using set-mapping function f can be calculated with $f(x)+c$. Figure 2-5 shows an example forest simulation forest.

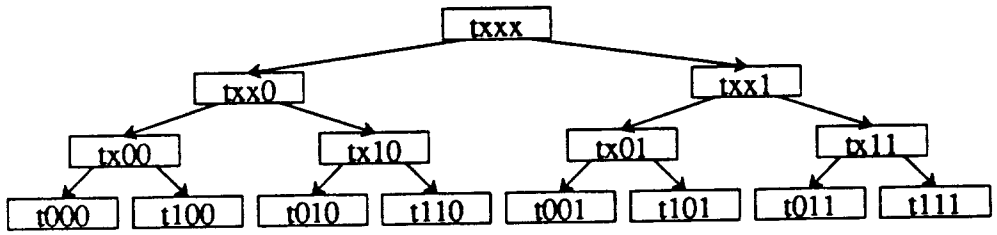


Figure 2-5. Forest Simulation Forest.

This figure displays the forest for caches of size 1, 2, 4, and 8 blocks. This forest contains only one tree, because the smallest cache contains only one block. This tree is a binary tree, because each cache in this example is twice as large as the next smaller cache. In this example I assume blocks are mapped to block frames with bit selection.

Each node holds the information for one block frame in a direct-mapped cache. The block at the root of the tree has no block number bits constrained, because a one-block direct-mapped cache can hold any block. This is illustrated with a *t* representing arbitrary high-order bits of the block number and three *x*'s representing *don't-cares* for the three low-order bits. The tags *txx0* and *txx1* in the nodes of level two indicate that the blocks that can reside in these nodes are constrained to have even and odd block numbers, respectively. Similar rules with more bits constrained apply to the rest of the levels.

Forest simulation works as follows and as is illustrated in Figure 2-6. On each reference, the algorithm selects the tree corresponding to the set of the reference in the smallest cache. Then it searches

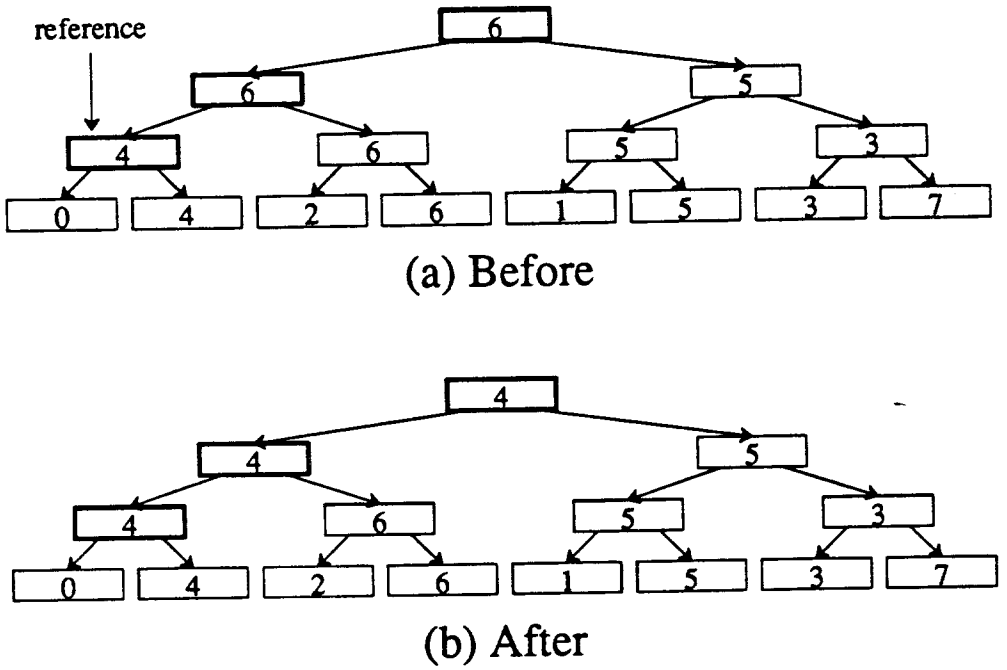


Figure 2-6. Forest Simulation Example.

The top tree (a) depicts the forest of Figure 2-5 after a series of references. Information in the tree tells us that block 6 is in a cache of size one block; blocks 6 and 5 are in a direct-mapped cache of size two; blocks 4, 6, 5 and 3 are in a direct-mapped cache of size four; and blocks 0 through 7 are in a direct-mapped cache of size eight.

Let the next reference be to block 4. A path from the root to a leaf is determined using the set-mapping function for each cache (here bit selection is assumed). A search begins at the root and stops when block 4 is found. All nodes encountered in the search that do not contain block 4 are modified to do so. The blocks in bold are examined to find block 4. Since block 4 is located at level 3, caches 1 and 2 miss and caches 3 and 4 hit.

The bottom tree (b) shows the tree after this reference as been processed. The nodes in bold now contain the referenced block.

integer L -- number of direct-mapped caches

-- set-mapping functions that obey set hierarchy,
-- i.e., $f_{i+1}(x)==f_{i+1}(y) \Longrightarrow f_i(x)==f_i(y)$ for i=1, L-1.

function $f_1(x)$, ..., $f_L(x)$

integer c_1, ..., c_L -- cache sizes (in blocks)

integer N -- number of references

-- distance counts so that $m(C(F=f_i, A=1) = 1 - \sum_{j=1}^{i} distance[i]/N$

integer distance[1:L]

integer forest[1:2*c_L] -- the forest
define block(x, i) = ($f_i(x) + c_i$) -- for accessing the forest

Figure 2-7. Forest Simulation Storage.

```
For each reference x {
        read(var x)
        N++
        -- FIND
        found = FALSE
        for i=1 to L or found {
                y = forest[block(x, i)]
                if (x==y)
                        found = TRUE
                        -- METRIC
                        distance[i]++
                else
                        -- UPDATE
                        forest[block(x, i)] = x
        }
}
```

Figure 2-8. Forest Simulation.

for the referenced block beginning at the root of the tree. The path of the search is determined by the set of the reference in each cache. Any time a node is encountered that does not contain the reference, the node is updated to contain it. The processing of a reference stops when the reference is found, or after a leaf node has been modified. If the reference is found at level i, a counter *distance[i]* must be updated.

Figures 2-7 and 2-8 show the pseudo-code for forest simulation. Forest simulation is efficient because it uses inclusion and direct-mapping. It uses inclusion in the same way as stack simulation, i.e., by ending the processing of a reference when it is found in a cache, regardless of how many larger caches are being simulated. Direct-mapping implies that a block can reside in only one block frame in a cache. Forest simulation benefits from direct-mapping by examining only that one block frame per cache. In contrast, a simulation of set-associative caches must often search more than one block frame per cache size of interest.

As with stack simulation, the exact storage required for a forest simulation of CPU caches is small relative to main memory sizes. The storage required is dominated by the size of the forest, which can be implemented in a heap of $2c_L$ nodes, where c_L is the number of blocks in the largest cache simulated (see Figure 2-7). The storage required for simulating direct-mapped caches with 32-byte blocks of sizes 128K, 256K, 512K and 1M byte, for example, is approximately 500K bytes, given node sizes of four to eight bytes.

Next I show the time used to process one reference in a forest simulation of L direct-mapped caches, C_i, is:

$$1 + \sum_{i=1}^{L-1} m_i + O(1) \qquad \text{Eq. 2-3.}$$

where m_i is the miss ratio of cache C_i and each iteration requires unit-time. The power of this analysis is limited, however, because several constant factors are difficult to calibrate.

The time to simulate each reference is determined by how many times the loop in Figure 2-8 is executed for each reference, plus a constant amount of overhead for reading trace addresses. Forest simulation executes one iteration per cache (level in the forest) up to a maximum of L levels. If one iteration requires unit-time, the execution time per reference is:

$$1*(1-m_1) + 2*(m_1-m_2) + \cdots + i*(m_{i-1}-m_i)$$
$$+ \cdots + L*(m_{L-1}-m_L) + L*m_L + O(1).$$

Rearranging terms yields:

$$1 - m_1 + \sum_{i=2}^{L} i(m_{i-1} - m_i) + Lm_L + O(1)$$

$$= 1 - m_1 + \sum_{i=2}^{L} i(m_{i-1}) - \sum_{i=2}^{L} i(m_i) + Lm_L + O(1).$$

This equation can be simplified by manipulating the third term, $\sum_{i=2}^{L} i(m_{i-1})$, so that the m_{i-1}'s are changed into m_i's. This manipulation changes the index variable from i to j, replaces j's with i+1's, simplifies, and changes summation bounds to yield:

$$\sum_{i=2}^{L} i(m_{i-1}) = \sum_{j=2}^{L} j(m_{j-1}) = \sum_{i+1=2}^{L} (i+1)(m_{i+1-1}) = \sum_{i=1}^{L-1} (i+1)(m_i)$$

$$= \sum_{i=2}^{L} (i+1)(m_i) + 2m_1 - (L+1)m_L.$$

Substituting this result back the time per reference equation produces:

$$1 - m_1 + (\sum_{i=2}^{L} (i+1)m_i + 2m_1 - (L+1)m_L) - \sum_{i=2}^{L} im_i + Lm_L + O(1),$$

which reduces to:

$$1 + m_1 + \sum_{i=2}^{L} 1 * m_i - m_L + O(1).$$

Readjusting summation limits yields a run-time per reference for forest simulation of:

$$1 + \sum_{i=1}^{L-1} m_i + O(1).$$

The miss ratios for L direct-mapped caches can also be computed with L *separate* or *concurrent* stack simulations of individual caches. In separate simulations, cache C_1 is simulated with all references, then cache C_2 is simulated with all references, and so forth, until cache C_L is simulated with all references. In concurrent simulation, all L caches are simulated at the same time with each reference processed by all the caches before the next reference is processed. Concurrent simulation is faster than separate simulation, but requires more storage. It is faster, because each trace address is read once rather than L times. It uses more storage, since blocks for all caches must be simultaneously resident. Since I care about run-time and not about storage, I consider only concurrent simulation further.

In concurrent simulation each address is read once, and unit processing is required for each level. The run-time per reference is, therefore:

$$L + O(1).$$

This time is greater than the time for forest simulation,

$$1 + \sum_{i=1}^{L-1} m_i + O(1),$$

for practical (not equal to one) miss ratios.

2.5. Simulating Set-Associative Caches without Inclusion

Stack and forest simulation will simulate a series of caches with one pass through an address trace. Both methods are "efficient," because they take advantage of inclusion. Since inclusion does not hold for caches of all sizes and associativities (see Theorem 1), algorithms using inclusion must constrain the series of caches simulated (see Figure 2-9). Here I describe an algorithm, which I call *all-associativity simulation*, that does not use inclusion, but can simulate set-associative caches with

the same block size, that do no prefetching, and use LRU replacement, with one pass over an address trace. With it, I can cover the design space of Figure 2-9 in 3 simulations (one per block size) instead of 15 runs of stack simulation. The algorithm described here permits the set-associative caches use of arbitrary set-mapping functions. A literature search revealed that a version of all-associativity simulation, where all set-mapping functions use bit selection, was developed by researchers at IBM [Matt70, Trai71][†].

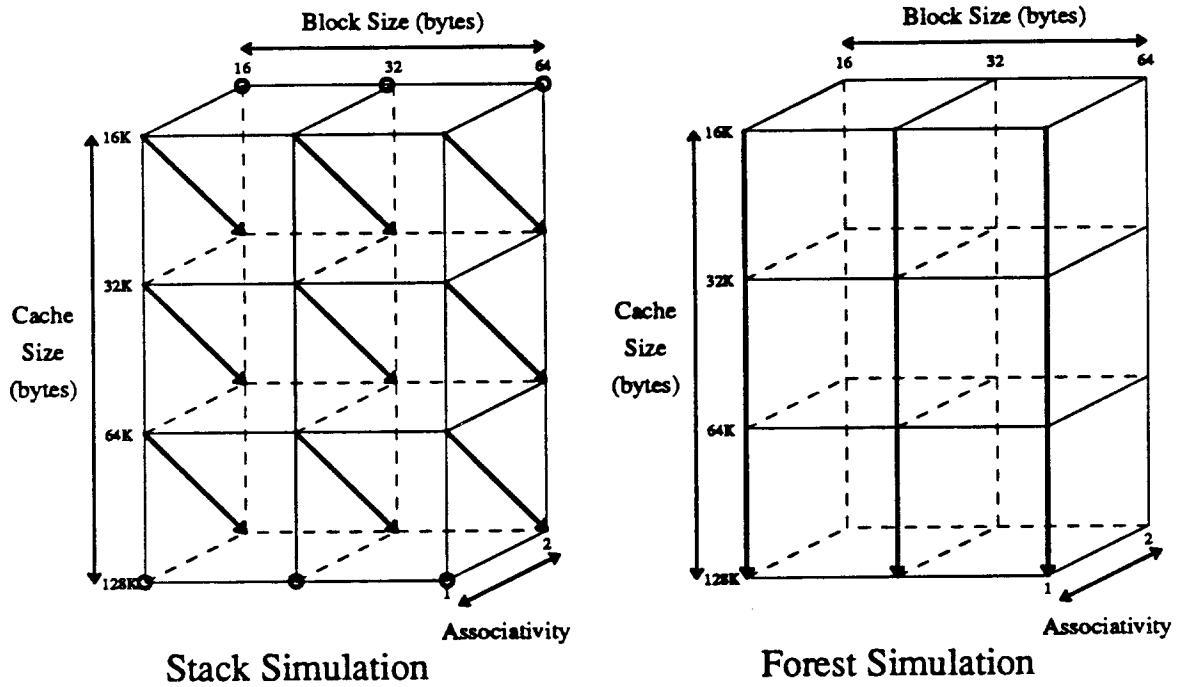


Figure 2-9. Simulating Cache Designs.

This figure displays a portion of the cache design space: cache size, block size and associativity. The solid lines on the left connects cache designs that can be simulated with a single stack simulation. Circles indicate simulations of single cache designs. The solid lines on the right connect cache designs that can be simulated with a single forest simulation.

Covering portions of the cache design space can require many simulations even though stack and forest simulation simulate several caches at a time. In this example, 15 stack simulations are needed, and 3 forest simulation cover half the space but can't simulate the rest of it. Alternatively, three all-associativity simulations (one per block size) cover the same space.

All-associativity simulation does not take advantage of inclusion, because inclusion does not hold for many groups of set-associative caches. For example: (a) direct-mapped and two-way set-associative caches of any size do not include any four-way set-associative caches, because the former have smaller associativities; (b) a four-way set-associative cache of c blocks does not include a direct-mapped cache of $c/2$ blocks even if both use bit selection, because $x \text{ rem } c/4 = y \text{ rem } c/4$ does not imply $x \text{ rem } c/2 = y \text{ rem } c/2$ (e.g., $x=0$ and $y=c/4$); and (c) it is not possible for a cache C_2 to include a different cache C_1 of the same size, because cache C_2 can never contain any blocks not in cache C_1 and still contain all the blocks of cache C_1.

I now develop all-associativity simulation from stack simulation through successive refinements. An all-associative simulation run can simulate caches that use different set-mapping functions, $f_i(x)$, and have different capacities. The same caches can be simulated with concurrent stack simulations. This approach requires a stack simulation for each different set-mapping function. For instance, if the

[†] Traiger and Slutz also describe how to simulate alternative caches with different block sizes.

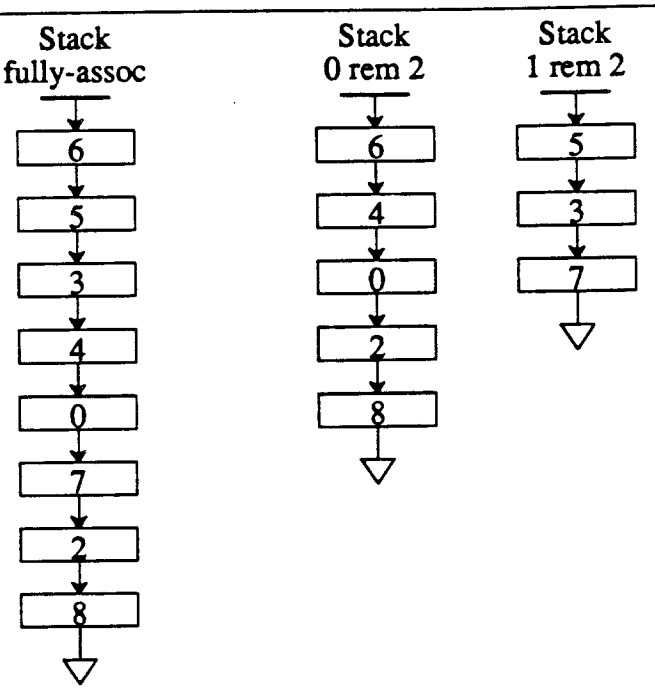


Figure 2-10. Concurrent Stack Simulation.

This figure displays how the stacks for caches with one (fully-associative) or two sets using bit selection ($f_1(x)=0$ and $f_2(x)=x$ rem 2) could look during a simulation. The stack for one set contains a list of all the block numbers recently referenced, listed from most-recently-referenced to least-recently-referenced. I call this stack a *fully-associative* stack, because it models fully-associative caches. The stacks for two sets contain similar lists for the even and odd block numbers. 2^L stacks are required to simulate with bit selection for 2^L sets. A block resides in a cache of c blocks with one set if and only if the block is in the fully-associative stack at a distance of less than or equal to c. A block resides in a cache of c blocks with two sets if and only if it is in the appropriate stack at a distance of less than or equal to $c/2$. A block resides in a cache of c blocks with 2^L sets if and only if it is in the appropriate stack at a distance of less than or equal to $c/2^L$.

caches to be simulated use bit selection with 0, 1, and 2 bits, the following stack simulations are sufficient: a stack simulation with one stack for caches with $f_1(x)=0$, a stack simulation with two stacks for caches with $f_2(x)=x$ rem 2, and a stack simulation with four stacks for caches with $f_3(x)=x$ rem 4. Figure 2-10 illustrates how the stacks for one and two sets with bit selection could look during concurrent stack simulation. In this example both sets of stacks contain the same nodes. In practice when stack sizes are bounded by the largest cache size of interest, the sets of stacks will (usually) contain slightly different nodes. For example, node 8 would be missing from stacks for two sets if caches of interest are restricted to eight total blocks (four per set). Nevertheless, many blocks will be shared by both caches, since similar caches have similar hit ratios.

Storage can be reduced in this simulation by allocating a single node per block and including in the node a *next* pointer field for each group of stacks being simulated. Figure 2-11 illustrates how the nodes of the single fully-associative stack can be linked with a second set of *next* pointers to form the stacks for caches with two sets.

While reducing storage is not important, this node sharing holds the key to reducing time. Observe that all the pointers in the stack point down. This is always the case for LRU replacement, because the order of two blocks in any stack is a function of references to those two blocks, independent of all other references. For example, stack 0 rem 2 in Figure 2-10 indicates that block 6 was referenced more recently than block 4. Block 6 must also be above block 4 in the fully-associative stack, because intermediate references to blocks 3 and 5 do not effect whether block 6 was referenced more recently than block 4. Since all pointers point down, the fully-associative stack contains all the information

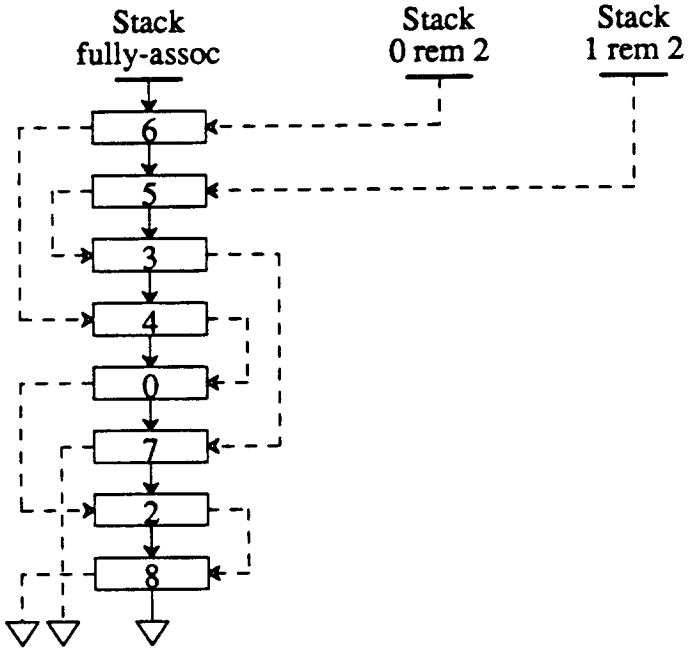


Figure 2-11. Concurrent Stack Simulation with Shared Storage.

This figure illustrates how a single set of nodes can be used to represent the stacks for caches using bit selection with one and two sets. A second *next* pointer field must be added to each node so that it can be linked into a second stack. The stacks for stack simulations with L different set-mapping functions can share one group of nodes if each node contains storage for L different *next* pointers. This reduces storage requirements with respect to using separate stacks, but does not reduce simulation time.

necessary to determine the order of nodes in all other groups of stacks. Thus, the stack for reference x with set-mapping function $f(x)$ can be constructed by finding all blocks y in the fully-associative stack where $f(y)=f(x)$ and listing these blocks in the same order as they are encountered in the fully-associative stack.

The goal of this research is to find stack distances, and hence miss ratios, however, not construct all the groups of stacks. The following algorithm computes stack distances for set-associative caches of different capacities and set-mapping functions f_1 through f_L directly from the fully-associative stack. When simulation completes, each counter *distance[i, k]* holds the number of references to stack distance k with set-mapping function f_i. For each reference to block number x {

Zero the L *total_above* counters.

Look at nodes y in the fully-associative stack until x is found or the stack exhausted. If $y=x$ {

Increment the L *total_above* counters, move x to the top of the stack, and increment *distance[i, total_above[i]]* for $i=1$ to L.

} else {

For $i=1$ to L, increment *total_above[i]* if $f_i(x)=f_i(y)$.

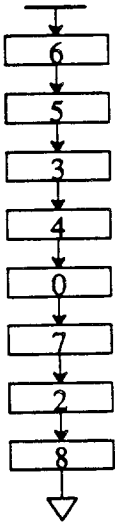
}

If the stack is exhausted without finding x, push x on the top of the stack.

}.

Figure 2-12 illustrates the algorithm operating on one reference. Figures 2-13 and 2-14 give pseudo-code for this algorithm.

All-associativity simulation can be improved further if I restrict the f_i's so that the set hierarchy condition holds. Recall that this condition is:



Stack fully-assoc	Block 2 found?	Fully-Assoc f(x) = 0 Same set?	Fully-Assoc f(x) = 0 Total_ above[1]	Two Sets f(x) = x rem 2 Same set?	Two Sets f(x) = x rem 2 Total_ above[2]	Four Sets f(x) = x rem 4 Same set?	Four Sets f(x) = x rem 4 Total_ above[3]
6	no	yes	1	yes	1	yes	1
5	no	yes	2	no	1	no	1
3	no	yes	3	no	1	no	1
4	no	yes	4	yes	2	no	1
0	no	yes	5	yes	3	no	1
7	no	yes	6	no	3	no	1
2	yes	yes	7	yes	4	yes	2
8	Stack Distance:		= 7		= 4		= 2

Figure 2-12. All-Associativity Simulation Example.

This figure illustrates how all-associativity simulation processes a reference to block 2 for caches with set-mapping functions $f_1(x) = 0$, $f_2(x) = x$ rem 2, and $f_3(x) = x$ rem 4. Counter *total_above[i]* always contains the number of blocks encountered so far stack $f_i(2)$, since block 2 is referenced. Each row of the figure shows that *total_above[i]* is incremented in response to block y in the stack if and only if $f_i(y) = f_i(2)$.

Processing stops when the reference is found (block 2). The stack distance of block 2 in a cache with set-mapping function f_i is *total_above[i]*. The stack distances found for block 2 are 7, 4, and 2, respectively.

```
integer L -- number of set-mapping functions
function f1(x), ..., fL(x) -- arbitrary set-mapping functions
integer N -- number of references
integer max_assoc -- maximum associativity for metrics
-- distance counts so that m(C(F=i, A=k)) = 1 - sum_{j=1}^{k} distance[i,k]/N
integer distance[1:L, 1:max_assoc]

define stacknode_type {
        integer block_number
        stacknode_type *next
}
stacknode_type *stack -- top of stack pointer
stacknode_type stacknodes[1:O(N*δ∞)] -- pool of dynamically linked stacknodes
```

Figure 2-13. All-Associativity Simulation Storage.

$f_{i+1}(x)=f_{i+1}(y)$ implies $f_i(x)=f_i(y)$.

In all-associativity simulation with arbitrary f_i's, it is necessary to know which of any two blocks are more recently referenced. Consequently, a total ordering of the previously referenced blocks must be

```
For each reference x {
        integer total_above[1:L]  -- distance counters for x

        for i=1 to L { total_above[i] = 0 }
        read(var x)
        N++

        -- FIND
        found = FALSE
        previous_node_pointer = NULL
        node_pointer = stack
        while ((NOT found) AND (node_pointer==NULL)) {

                y = node_pointer->block_number

                if (x==y) {
                        found = TRUE
                        for i=1 to L { total_above[i]++ }
                }
                else {
                        for i=1 to L {
                                if (f_i(x)==f_i(y)) total_above[i]++
                        }
                        previous_node_pointer = node_pointer
                        node_pointer = node_pointer->next
                }
        }

        -- METRIC
        if (found) {
                for i=1 to L {
                        -- References to distances beyond max_assoc are implicit misses.
                        if (total_above[i] ≤ max_assoc) distance[i, total_above[i]]++
                }
        }

        -- If was found, move the stack node of x to the top of the stack.
        -- Otherwise, store x in a new stacknode and move it to the top of the stack.

        UPDATE(x, found, previous_node_pointer, node_pointer)
}
```

Figure 2-14. All-Associativity Simulation.

maintained with a fully-associative LRU stack. Since the set hierarchy condition also implies:

$$f_1(x) \neq f_1(y) \text{ implies } f_i(x) \neq f_i(y) \text{ for } i=1,L,$$

two blocks in different f_1-stacks will never be compared. This means all-associativity simulation with set hierarchy need only maintain the LRU stacks for each element in the image of f_1. Simulating with multiple stacks is faster than simulating with one, because the average number of active blocks one must look through to find a block is smaller, since active blocks are spread across many stacks. This reduction is significant since, the number of stacks for practical CPU cache simulations is often greater than 100. The number of stacks used in a simulation of the VAX-11/780's cache, for example, is 512.

Another benefit of set-hierarchy is that a simulation need not examine $f_i(x)=f_i(y)$ for $i=L$ down to 1, since $f_{i+1}(x)=f_{i+1}(y)$ implies $f_i(x)=f_i(y)$. Instead of iterating through all L set-mappings, one can begin with f_L and stop as soon as $f_i(x)=f_i(y)$. For instance, if x and y are in the same set in the largest cache simulated, i.e., $f_L(x)=f_L(y)$, the number of iterations is reduced from L to one. Additional time can be saved if one increments *above[i]* only for the largest i for which x and y map to the same set, rather than incrementing *total_above[i]* for each i where $f_i(x)=f_i(y)$. When x is found or the stack

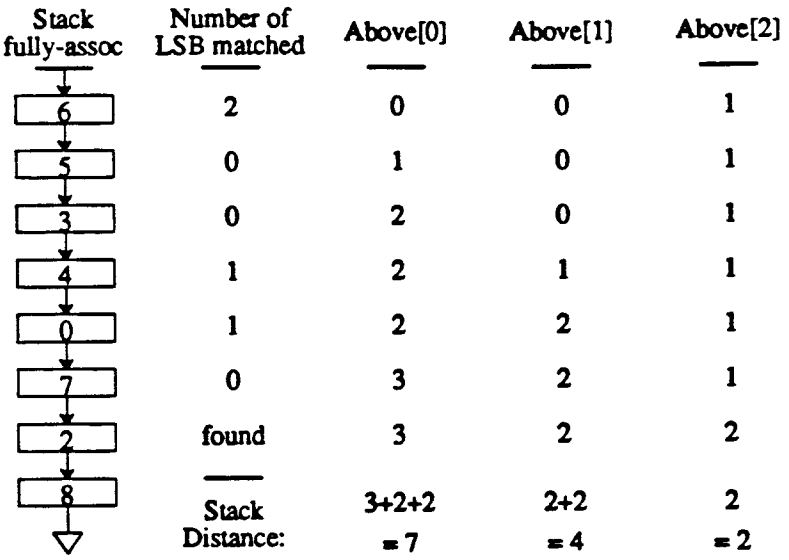


Figure 2-15. All-Associativity Simulation with Set Hierarchy Example.

This figure illustrates how all-associativity simulation with set hierarchy processes a reference to block 2 by scanning the stack until block 2 is found (or the stack is exhausted). For each block before the reference is found: (a) The algorithm calculates the largest set-mapping function, f_i, for which the reference and the stack node are in the same set. For bit selection, the calculation reduces to computing the number of least-significant bits that match between the block numbers of the reference and the stack node. (b) it increments $above[i]$. Once the reference is found, $above[L]$ is incremented, the reference's stack distance with set-mapping function f_i is $\sum_{k=i}^{L} above[k]$.

integer L -- number of set-mapping functions

-- set-mapping functions that obey set hierarchy,
-- i.e., $f_{i+1}(x)=f_{i+1}(y) \Rightarrow f_i(x)=f_i(y)$ for i=1, L-1.

function $f_1(x)$, ..., $f_L(x)$

integer number_of_stacks -- number of sets induced by $f_1(x)$

integer N -- number of references

integer max_assoc -- maximum associativity for metrics

-- distance counts so that $m(C(F=i, A=k)) = 1 - \sum_{j=1}^{k} distance[i,k]/N$

integer distance[1:L, 1:max_assoc]

```
define stacknode_type {
        integer block_number
        stacknode_type *next
}
```

stacknode_type *stack[0:number of stacks-1] -- top of stack pointers

stacknode_type stacknodes[1:O(N*δ_{max})] -- pool of dynamically linked stacknodes

Figure 2-16. All-Associativity Storage w/ Set Hierarchy.

```
For each reference x {
        integer above[1:L]  -- distance counters for x

        for i=1 to L { above[i] = 0 }
        read(var x)
        N++
        stack_number = f1(x)

        -- FIND
        found = FALSE
        previous_node_pointer = NULL
        node_pointer = stack[stack_number]

        while ((NOT found) AND (node_pointer==NULL)) {

                y = node_pointer->block_number

                if (x==y) {
                        found = TRUE
                        above[L]++
                }
                else {
                        match = FALSE
                        for i=L down to 1 or match {
                                if (fi(x)==fi(y)) {
                                        match = TRUE
                                        above[i]++
                                }
                        }
                        previous_node_pointer = node_pointer
                        node_pointer = node_pointer->next
                }
        }

        -- METRIC
        if (found) {
                total_above = 0
                for i=L down to 1 {
                        total_above = total_above + above[i]
                        -- References to distances beyond max_assoc are implicit misses.
                        if (total_above ≤ max_assoc)  distance[i, total_above]++
                }
        }

        -- If was found, move the stack node of x to the top of its stack.
        -- Otherwise, store x in a new stacknode and move it to the top of the stack.

        UPDATE(x, stack_number, found, previous_node_pointer, node_pointer)
}
```

Figure 2-17. All-Associativity Simulation w/ Set Hierarchy.

exhausted,

$$total\_above[i] = \sum_{k=i}^{L} above[k].$$

Figure 2-15 gives an example of all-associativity simulation using set hierarchy. Figures 2-16 and 2-17 give the pseudo-code for this improved algorithm.

All-associativity simulation can be made to run even faster in practice if the f_i's are all bit selection. Bit selection set-mapping functions make it easy to compute the largest i for which x and y map to the same set. The computation reduces to finding the minimum of L and the number of least

significant bits that match between x and y.

I have defined all-associativity simulation for set-associative caches that use LRU replacement. I now show that it does not work with two other commonly-implemented replacement algorithms, FIFO and RANDOM. All-associativity simulation does not work with FIFO replacement, because all-associativity simulation is based on stack simulation, and FIFO is not a stack algorithm (see Section 2.2.2).

Figure 2-18 shows by example that all-associativity simulation does not work with RANDOM replacement even though RANDOM is a stack algorithm. The example illustrates the following general problem. Any replacement algorithm may reorder blocks in the set of a reference between the top of stack and the original position of the reference, so long as no blocks other than the reference move up. In all-associativity simulation, multiple set-mapping functions are concurrently simulated. Therefore, some blocks can be in the set of a reference with one set-mapping function and not in the set of a reference with another set-mapping function. Incorrect behavior occurs any time blocks not in the set of the reference are reordered. LRU prevents such blocks from being reordered by never changing the order of unreferenced blocks.

While I have shown that all-associativity simulation fails with FIFO and RANDOM, I have not shown that all-associativity simulation fails with all replacement algorithms other than LRU. One way to show this is to prove the following. Consider an all-associativity simulation with two set-mapping functions, $f_1 \neq f_2$. (Recall that all-associativity simulation reduces to stack simulation if only one set-mapping function is used.) The stacks in all-associativity simulation are updated incorrectly in response to a reference x if blocks not in $f_1(x)$ or not in $f_2(x)$ are reordered. While I have not done it, one can demonstrate that LRU replacement is necessary for all-associativity simulation by showing that any replacement algorithm which obeys the above constraint never reorders any unreferenced blocks, and is therefore equivalent to LRU.

The storage for all-associativity simulation is dominated by storage for the stack nodes (see Figure 2-16). Like unbounded stack simulation, the storage required is proportional to the number of unique blocks in a trace, $N\delta_\infty$. Even for a long trace, however, the storage required is small relative to modern main memory sizes. A simulation of 10 million references with 200 thousand unique blocks requires only 1.6M bytes of storage if it uses two words per block. While stacks in stack simulation can be bounded by the largest associativity of interest, stacks in all-associativity simulation cannot be bounded, because these stacks are used to construct stacks for other set-mapping functions. Consider a fully-associative stack and stacks for even and odd blocks. The fully-associative stack cannot be bounded, because the first odd block can reside at an arbitrary large distance in the fully-associative stack.

The run-time (per reference) of all-associativity simulation with set hierarchy centers around how many times the "while" loop is executed (see Figure 2-17). Let δ_k be the probability that a reference is found at stack depth k, and let δ_∞ be the probability that a reference is not found. References at stack distance k are found in k iterations. References at stack distance ∞ are found by looking through the entire stack[†]. The size of the stack is equal to the number of distinct blocks previously referenced, which is $O(N\delta_\infty)$, where N is the number of blocks in the address trace. On each iteration in all-associativity simulation, a stack node must be compared to the reference to see if they are the same. If not, additional work is required to find the maximum i for which the reference and the stack node are in the same set. Let the average amount of this extra work be called *match_compute*. Whenever a reference is found at distance k, unit work must be done on k iterations and *match_compute* work on all but the last iteration. In addition, each reference must be read from a trace file, L *above* counters initialized and summed to form the stack distances. I gather the per-reference overhead in $O(1)$. Thus, time to process a reference is of order:

[†] As discussed in Section 2.2.4, a hash table can be used to suppress the search for previously unreferenced blocks.

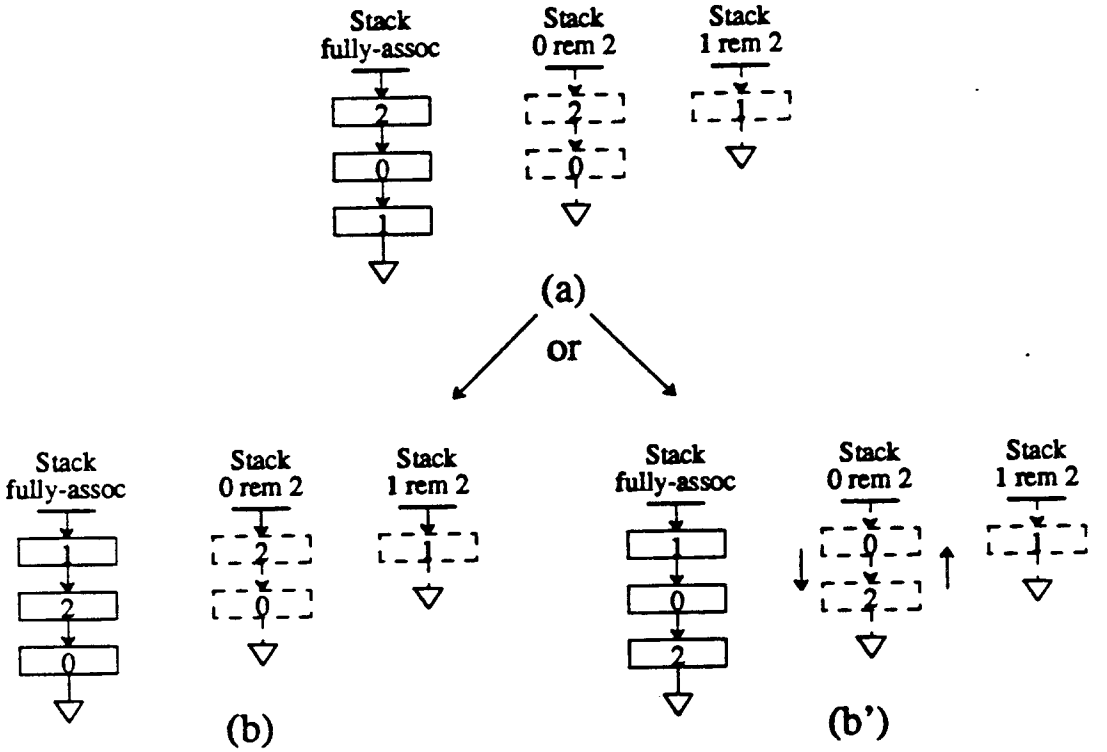


Figure 2-18. Random Replacement Does Not Work.

This figure shows that all-associativity simulation does not work with RANDOM replacement. Part (a) illustrates a fully-associative stack after a series of references (left), and the pair of stacks for even and odd blocks implied by the fully-associative stack (right). Among other things, the stacks imply that a two-block fully-associative cache contains blocks 0 and 2, and a two-block direct-mapped cache contains blocks 1 and 2.

Let block 1 be referenced. RANDOM replacement in the two-block fully associative cache requires that block 0 or block 2 be replaced with equal probability. Part (b) shows block 0 replaced, while part (b') shows block 2 replaced. The stacks in part (b) are consistent, since RANDOM replacement coincidentally replaces the least recently used block.

The state of the fully-associative stack in part (b'), however, implies that block 0 is in the two-block direct-mapped cache. The state of the stacks is inconsistent, since block 0 was not originally in the two-block direct-mapped cache, block 0 was not referenced, and prefetching is not allowed. Therefore the fully-associative stack under RANDOM replacement cannot be used to infer the positions of blocks in the even and odd stacks, which demonstrates that all-associativity simulation does not work with RANDOM replacement.

$$\sum_{k=1}^{\bar{\infty}} \delta_k * [k + (k-1) * match\_compute] + \delta_{\infty} * O(N\delta_{\infty}) * [1 + match\_compute] + O(L) + O(1). \qquad \text{Eq. 2-4.}$$

The first term is the time to process previously referenced blocks; the second is for previously unreferenced blocks; the third and final terms are for manipulating counters and reading the reference, respectively.

To see how this run-time compares with stack simulation, let us assume the all set-mapping functions are bit-selection. For this to be possible with a 32-bit address, L must be less than 32. If the low-order bits of block numbers for recently-referenced blocks are independent and equally likely to be zero or one, then the expected number of least-significant bits that match is less than 1 (1/2 + 1/4 + 1/8 + ...). Since the loop computing match iterates until a mismatch is found, the expected number of iterations is two[†]. Substituting two for *match_compute* yields:

[†] On some machines *match* can be computed with a fixed number of instructions (e.g., applying the VAX-11 instruction FIND-FIRST-SET to the exclusive-OR of the addresses being compared. [Digi81]).

$$\sum_{k=1}^{\infty} \delta_k * [3k - 2] + 3*O(N\delta_{\infty})*\delta_{\infty} + O(32) + O(1), \qquad \text{Eq. 2-5.}$$

which should not be more than three times greater than the time for stack simulation (Equation 2-1):

$$\sum_{k=1}^{\infty} k\delta_k + O(N\delta_{\infty})*\delta_{\infty} + O(1).$$

In practice the relative difference in run times should be smaller, because I expect that the $O(32)$ term to be small compared to other terms, δ_1 to be near one (the direct-mapped hit ratio near one) often saving any *match_compute* overhead, and the per-reference overhead $O(1)$ to be relatively large.

2.6. Comparing Actual Simulation Times

Here I compare the simulation times of implementations of stack, forest, and all-associativity simulation. While the exact quantitative results of this section do not necessarily apply to other implementations, there is no reason to believe that gross comparisons do not generalize. The advantage of this data over the run-time analysis presented earlier is that these results apply to at least one set of implementations of these algorithms.

I have implemented stack, forest and all-associativity simulation in C under UNIX 4.3 BSD. Stack and forest simulation were added to a general cache simulator, called *DineroIII* [Hill85]. DineroIII originally contained 1250 C statements, as measured by the number of source lines containing a semicolon or closing brace. Adding stack simulation increased total code size by 150 statements, adding forest simulation, 220 statements. Stack simulation is implemented using linked lists and without using a hash table to detect previously unreferenced blocks (see Section 2.2.4). The forest simulation implementation restricts the set-mapping functions to be the block number modulo the cache size in blocks, a generalization of bit selection. I implemented all-associativity simulation in a separate program, called *Tycho*, containing 800 C statements and having far fewer options than DineroIII. Tycho restricts the set-mapping functions to be bit selection. My implementations of these algorithms are available to interested researchers free of charge.

I estimate simulation time with the elapsed virtual time (user plus system) returned by the UNIX 4.3 BSD system call *getrusage* on an otherwise unloaded Sun-3/75 with 8M of memory and no local disk. Trace data is read from a file server via an ethernet. I give the results for four traces from four different architectures, described in Table 2-1, despite finding that results are fairly insensitive to program traces. All caches simulated have 32-byte blocks, do no prefetching, use LRU replacement, are mixed (data and instruction cached together), and use bit selection.

I begin by verifying that implementations of the three algorithms have similar run-times for simulating a single cache, using two methods. First, I ran each implementation using a trace of 1 million identical addresses so that all references, expect the first, hit at distance one. Results show that the elapsed virtual times of forest and stack simulation differ by 0.1 percent, while all-associativity simulation is 3 percent faster. All-associativity simulation is faster, because it is implemented in a different program, Tycho. It is not surprising that Tycho is slightly faster than DineroIII, because DineroIII is a general cache simulator. Even though DineroIII's additional features are not used in these simulation runs, DineroIII uses some execution time to fall through the *if* statements that guard the additional features.

Second, I compare the algorithms simulating a 16K-byte direct-mapped cache with each of four traces (See Table 2-1). Figure 2-19 displays these results, which are similar to those above. In addition, Figure 2-20 shows that a stack and an all-associativity of a single 16K-byte four-way set-associative cache are also comparable.

Since my implementations of these algorithms have similar run-times for simulating single caches, the comparisons of multiple cache simulations that follow are meaningful, because I know that simulation time differences are not due to per reference overheads.

Trace Name	Architecture; Operating System	Comments
mvs	IBM 370; MVS	System calls from an Amdahl standard MVS workload [Smit85].
synapse_devel	Single-68000 Synapse N+1; Synthesis	User and system trace of a synthetic development workload consisting of editing, compiling, linking and executing a collection of programs, as well as executing common system utilities (directory list, etc).
mul2.000	VAX-11; VMS	User and system trace of a two-job multiprogrammed workload gathered with microcode modifications [Agar86].
unoptupas	MIPS; Unix BSD	User trace of unoptimized version of the Pascal front-end for the MIPS system compiling a TLB simulator.

Table 2-1. Program Address Traces.

This table describes the four traces from four architectures used to evaluate stack, forest and all-associativity simulation. Results are fairly insensitive to program traces. More details can be found in [Agar86, Smit85]

Next I verify my expectation that forest simulation is faster than stack and all-associativity simulation for simulating a series of direct-mapped caches. Figure 2-21 shows the running times for one run of the three algorithms simulating 16K, 32K, 64K, and 128K-byte direct-mapped caches. Forest and all-associativity simulation require only one run to simulate the four caches, while stack simulation uses four runs. Forest simulation is between 1.3 and 4.4 percent faster than all-associativity simulation. A single stack simulation run cannot simulate more than one direct-mapped cache, instead, one run per cache is required. Lines labeled "S/4" show the stack simulation times required to simulate the 128K-byte direct-mapped cache. The stack simulation time for the series of four direct-mapped caches is at least four times as great.

The reason that the simulation times for single runs of the three algorithms are similar for this series of direct-mapped caches is that the miss ratios are relatively small, less than 5 percent. For example, the miss ratios for trace *mvs* for direct-mapped caches from 16K to 128K-bytes are 7.0, 4.6, 2.6, and 1.7 percent. On each direct-mapped cache hit, the reference is found at the top of the stack by each algorithm, and therefore simulation times have little opportunity to differ. Consequently I re-ran the algorithms on smaller direct-mapped caches. The *mvs* miss ratios for these caches, from 1K to 8K-bytes, are 20.1, 16.9, 13.3, and 9.8 percent. As Figure 2-22 indicates, the simulation time differences did indeed increase. Forest simulation is between 8 and 24 percent faster than all-associativity simulation.

I now verify my expectation that all-associativity simulation is faster than stack simulation for simulating caches when size and associativity are independently varied, because all-associativity simulation needs fewer simulations to do the job. I choose to simulate the twelve caches of size 16K, 32K, 64K and 128K-bytes, and associativity one, two and four, because these sizes and associativities are typical of CPU caches. Figure 2-23 illustrates the six stack simulations necessary to cover this region of the design space. The run-times in Figure 2-24 show that the time for the all-associativity simulation is comparable to that of one of the six stack simulations required, and hence, all-associativity simulation covers this region of the cache design space about six times faster.

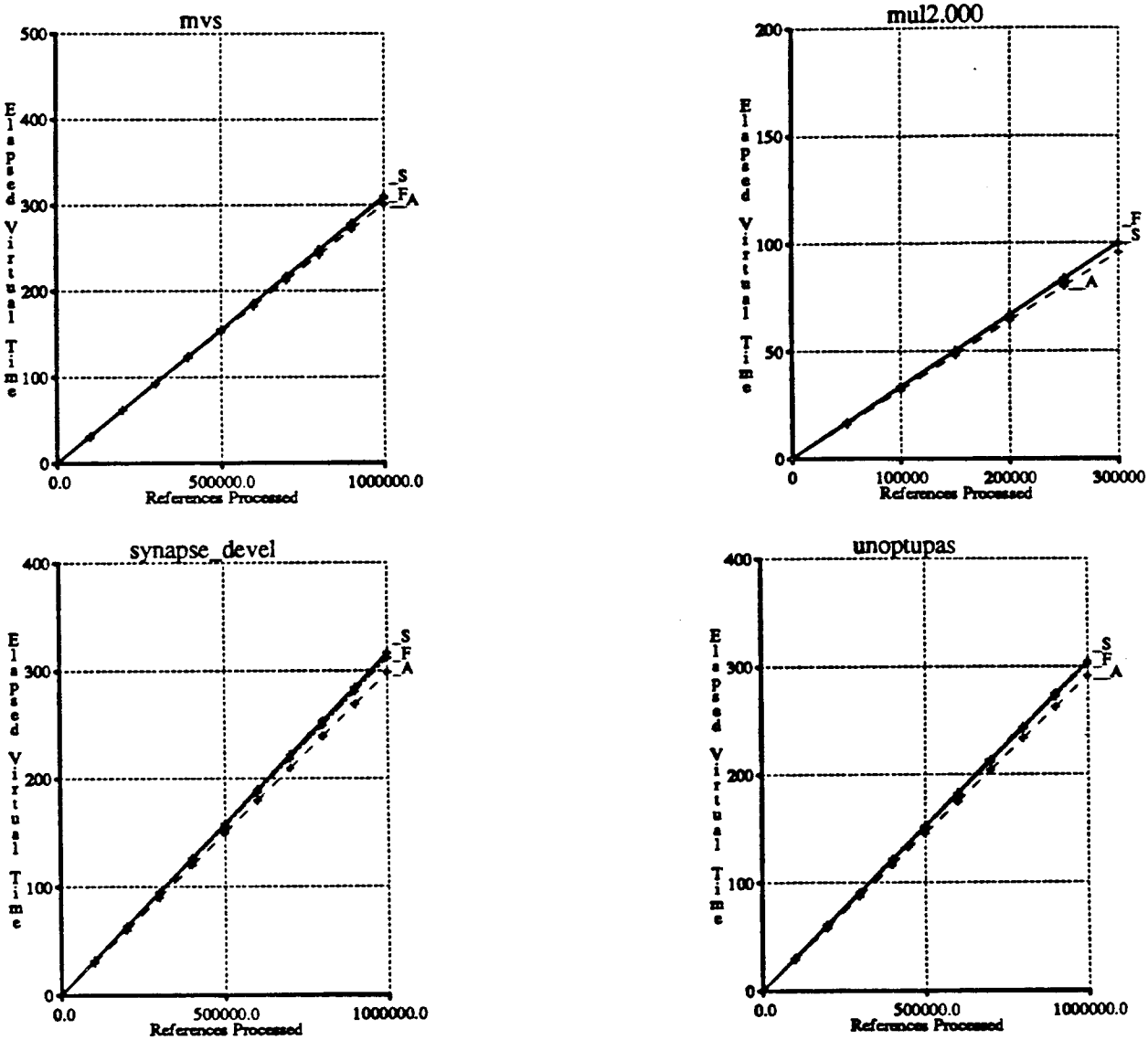


Figure 2-19. Single Direct-Mapped Cache Run-Times.

This figure displays the elapsed virtual time in seconds for a stack ("S"), a forest ("F") and an all-associativity simulation ("A") of a 16K-byte direct-mapped cache with each of the four traces, revealing that all three algorithms perform similarly. Note that trace *mul2.000* is one-third the length of the other three traces.

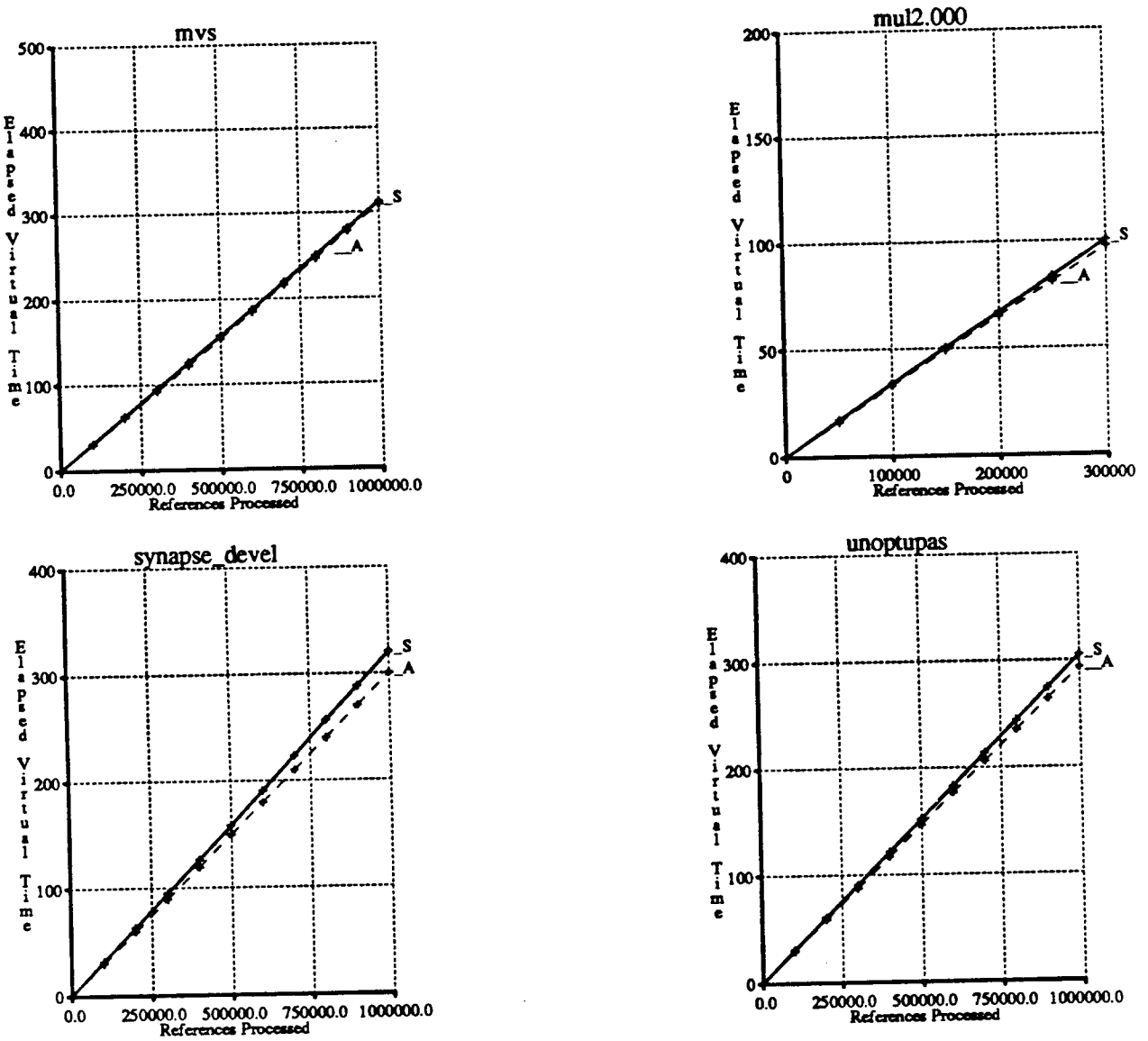


Figure 2-20. Single Set-Associative Cache Run-Times.

This figure displays the elapsed virtual time in seconds for a stack ("S") and an all-associativity simulation ("A") of a 16K-byte four-way set-associative cache with each of the four traces, showing both algorithms perform similarly. A forest simulation is not included, because forest simulation operates on direct-mapped caches only.

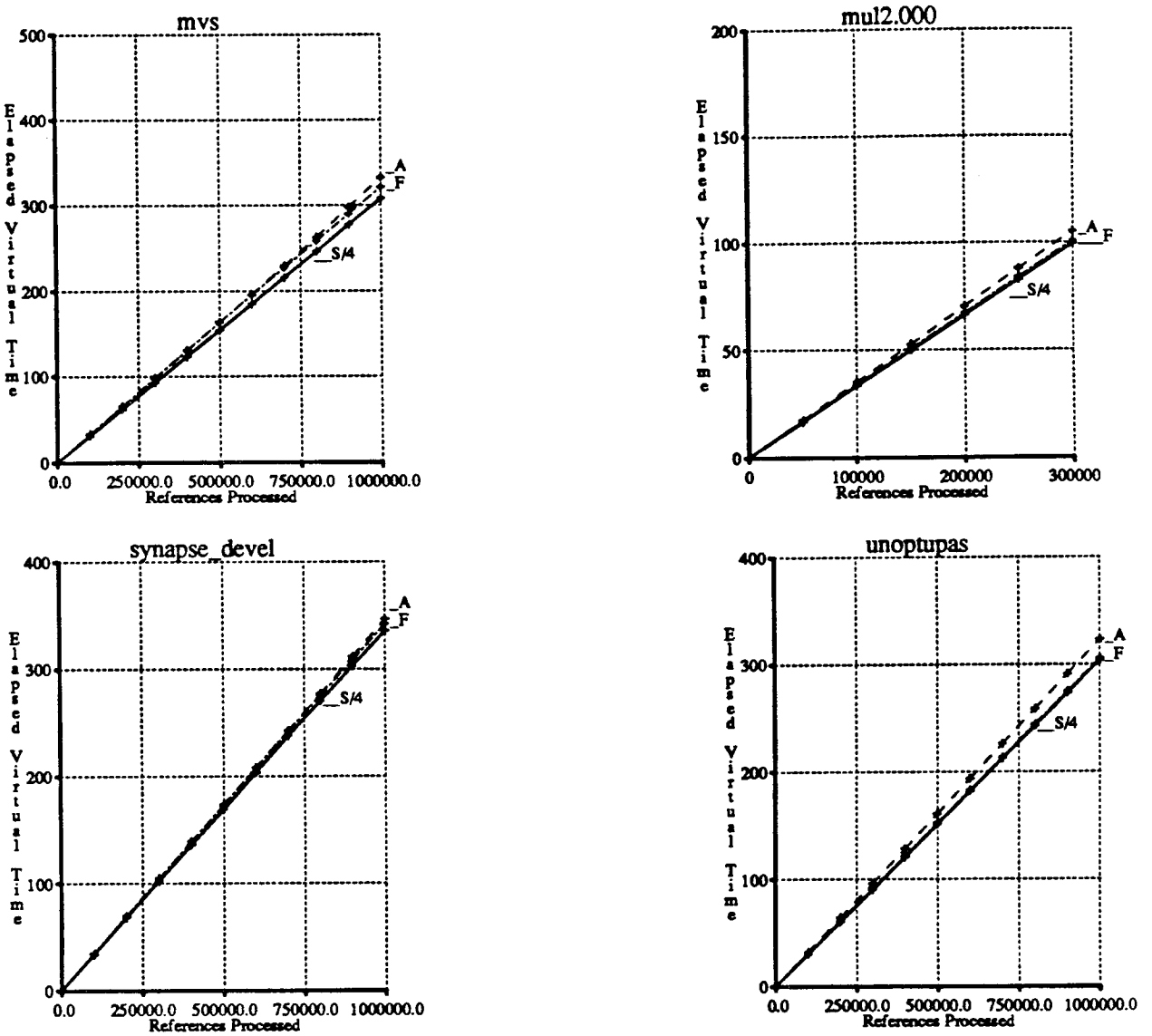


Figure 2-21. Run-Times for a Series of Direct-Mapped Caches.

This figure displays the elapsed virtual time in seconds for one run of stack ("S/4"), forest ("F") and all-associativity simulation ("A") evaluating 16K, 32K, 64K, and 128K-byte direct-mapped caches with each of the four traces. Forest and all-associativity simulation require one run to simulate the caches, whereas stack simulation needs four runs, one run per cache size. The forest simulation performs slightly faster than all-associativity simulation, and considerably faster than stack simulation for this series of direct-mapped caches. Stack simulation is inferior, because it requires four separate simulation runs. Lines "S/4" show the fastest of these four runs.

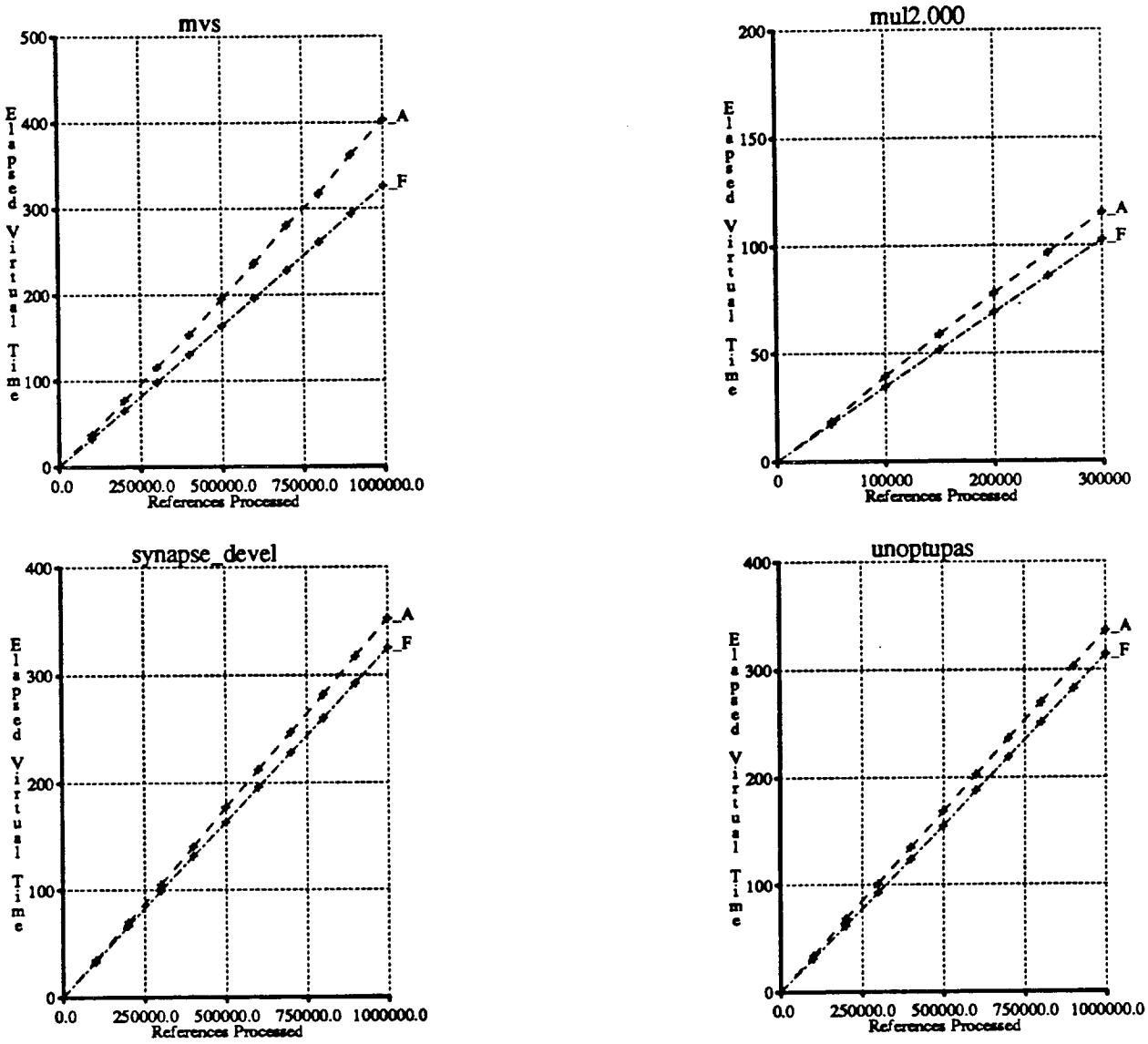


Figure 2-22. **Run-Times for Small Direct-Mapped Caches.**

Similar to Figure 2-21, this figure displays the elapsed virtual time in seconds for a forest and an all-associativity simulation of 1K, 2K, 4K, 8K-byte direct-mapped caches. Simulation time differences are larger here than in Figure 2-21, because the miss ratio here are larger.

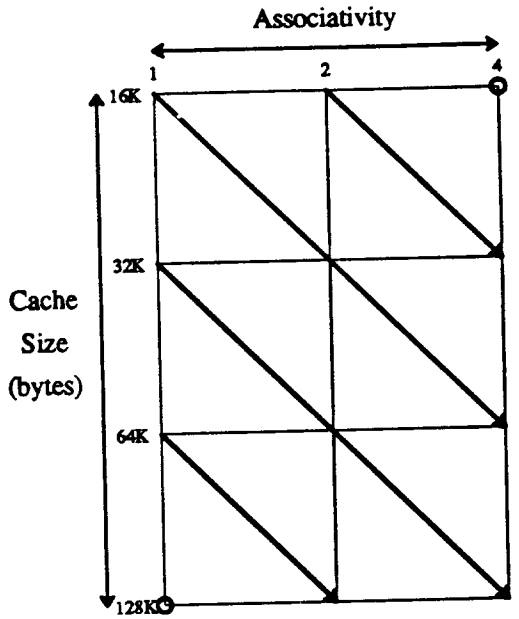


Figure 2-23. Simulating a Design Region with Stack Simulation.

This figure displays the stack simulations necessary to simulate the twelve caches of size 16K, 32K, 64K and 128K-bytes, and associativity one, two and four. Bold lines connect caches that can be simulated with a single stack simulation. Bold circles indicate the simulation of a single cache. A stack simulation is required for each different number of sets. Six separate simulations are required:

- 128 sets (16K-byte with four-way)
- 256 sets (16K-byte with two-way and 32K-byte with four-way)
- 512 sets (16K-byte with direct-mapped, 32K-byte with two-way and 64K-byte with four-way)
- 1K sets (32K-byte with direct-mapped, 64K-byte with two-way and 128K-byte with four-way)
- 2K sets (64K-byte with direct-mapped and 128K-byte with two-way)
- 4K sets (128K-byte with direct-mapped).

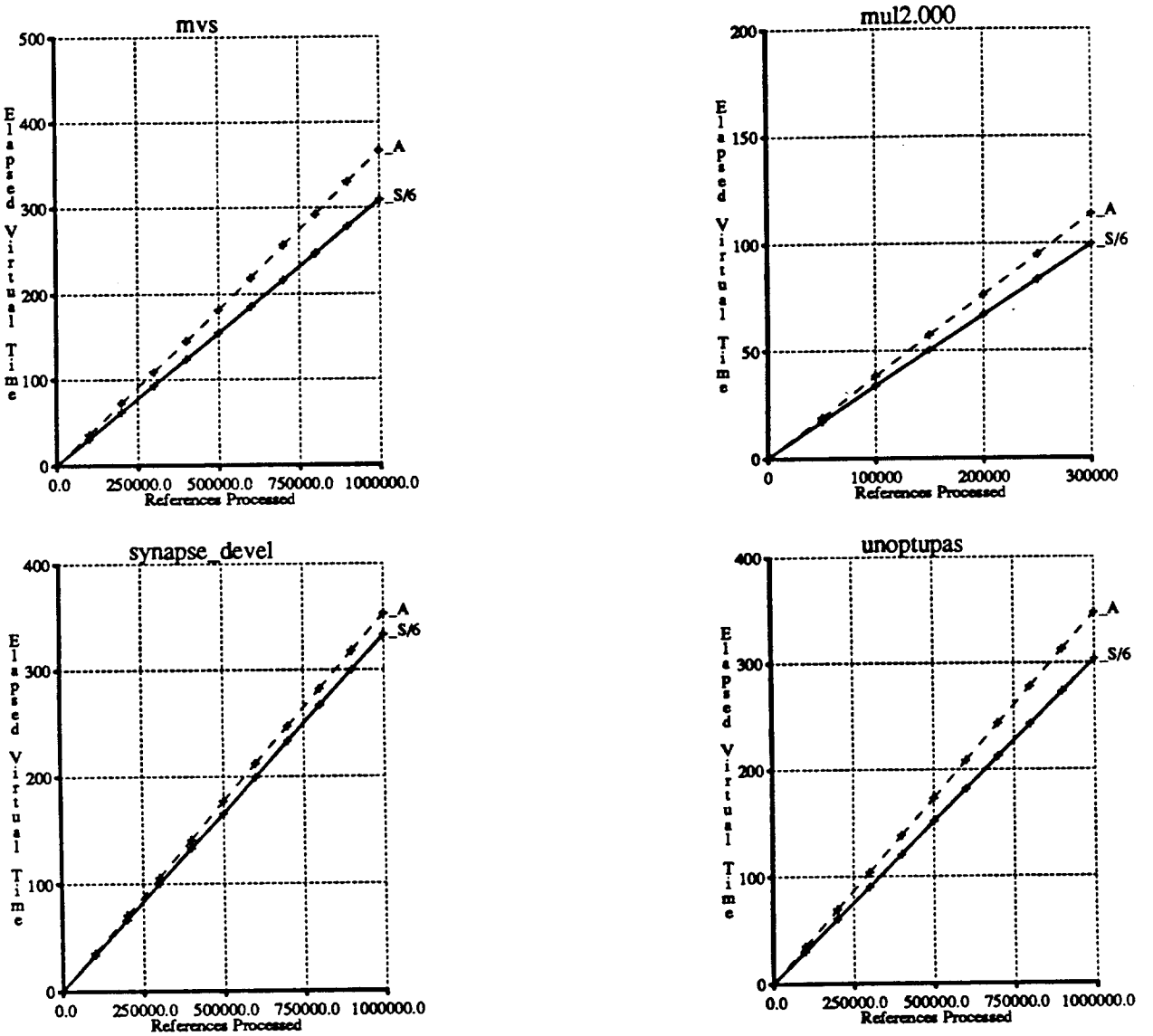


Figure 2-24. Run-Times for Simulating Twelve Similar Caches.

This figure shows the elapsed virtual time in seconds for one one of stack ("S/6") and all-associativity simulation ("A") evaluating caches of 16K, 32K, 64K and 128K bytes with associativity one, two, and four with each of the four traces. While all-associativity simulation requires only one run to evaluate the twelve caches, stack simulation need six runs. The stack simulation lines illustrate the time for the fastest of the six stack simulation times. Consequently, the total stack simulation time is at least six times greater than the time displayed by this line. All-associativity simulation is clearly superior to stack simulation for simulating a design region where cache size and associativity are independently varied.

I conclude by examining how well the run-time analysis done for stack simulation in Section 2.2.3, for forest simulation in Section 2.4 and for all-associativity simulation in Section 2.5 predicts actual simulation times. I illustrate this comparison with simulation times from trace *mvs*.

All three algorithms simulate single direct-mapped caches in $1.00 + O(1)$ per reference.

Forest simulation requires the following run-time per reference for L direct-mapped caches (Equation 2-3):

$$1 + \sum_{i=1}^{L-1} m_i + O(1).$$

This equation reduces to the following when four caches are simulated:

$$1 + \sum_{i=1}^{3} m_i + O(1).$$

Since the miss ratios with *mvs* are 7.0, 4.6, 2.6, and 1.7 percent for 16K to 128K-byte direct-mapped caches, the predicted run-time per reference of forest simulation with these caches is $1.14 + O(1)$.

Bounded stack simulation requires the following run-time per reference (Equation 2-2):

$$\sum_{k=1}^{kmax} k\,\delta_k + kmax^* \sum_{k=kmax+1}^{\infty} \delta_k + kmax^*\,\delta_\infty + O(1).$$

This equation reduces to the following for caches up to four-way set-associative:

$$\sum_{k=1}^{4} k\,\delta_k + 4^* [1 - \sum_{k=1}^{4} \delta_k] + O(1).$$

The δ's for a simulation of a four-way set-associative, 16K-byte cache with trace *mvs* are: $\delta_1 = 0.8661$, $\delta_2 = 0.0502$, $\delta_3 = 0.0200$ and $\delta_4 = 0.0128$. Consequently the predicted run-time per reference for that stack simulation is $1.28 + O(1)$.

All-associativity simulation requires the following run-time per reference (Equation 2-4):

$$\sum_{k=1}^{\infty} \delta_k * [k + (k-1)*match\_compute] + \delta_\infty * O(N\,\delta_\infty) * [1+match\_compute] + O(L) + O(1).$$

This equation reduces to the following for simulating four different numbers of sets provided that I assume *match_compute* is 2 (see discussion preceding Equation 2-5):

$$\sum_{k=1}^{\infty} \delta_k * [3*k - 2] + 3*\delta_\infty * O(N\,\delta_\infty) + O(4) + O(1).$$

An all-associative simulation of *mvs* with direct-mapped caches of size 16K through 128K bytes has $\delta_1 = 0.9305$, $\delta_2 = 0.0336$, $\delta_3 = 0.0198$ and $\delta_4 = 0.0073$; therefore its run-time per reference is:

$$1.28 + \sum_{k=5}^{\infty} \delta_k * [3*k - 2] + 3*\delta_\infty * O(N\,\delta_\infty) + O(4) + O(1).$$

I can compute a lower bound on the run-time by assuming that all blocks not found at stack distances one through four are found at distance five, and that the $O(4)$ term can be ignored. These assumptions yield a run-time per reference of at least $1.39 + O(1)$.

Table 2-2 displays the run-time predictions calculated above versus actual simulation times for the *mvs* trace. Results show that:

(1) Run-time predictions correctly order different runs of the same algorithm, but not runs of different algorithms.

(2) All simulations of a single cache, with expected run-time $1.00 + O(1)$, require about 300 µs/reference on a Sun-3/75 (5 minutes/1M-references).

(3) All simulations are dominated by the constant term $O(1)$. A change that increases a run-time prediction from $1.00 + O(1)$ to $1.10 + O(1)$ leads to a simulation time increase of about one percent.

2.7. Conclusions

Mattson et al. demonstrated when inclusion holds and how to take advantage of it for simulating alternative single-level cache designs having the same number of sets [Matt70]. I examine inclusion and simulation algorithms for caches having differing numbers of sets, because alternative CPU cache designs rarely have the same number of sets.

Since I find inclusion usually holds for direct-mapped CPU caches (e.g., those that use bit selection), I describe an algorithm for rapidly simulating alternative direct-mapped caches that uses inclusion. This algorithm, called *forest simulation*, is considerably faster than stack simulation for

Algorithm	Cache Size (bytes)	Degree of Associativity	MVS Runtime/ Reference	MVS s/1M-Reference	Normalized To DM-16K
Forest	16K	1-way	1.00 + O(1)	307.6	1.0
	16K to 128K	1-way	1.14 + O(1)	321.0	1.044
	1K to 8K	1-way	1.51 + O(1)	326.1	1.060
Stack	16K	1-way	1.00 + O(1)	309.2	1.0
	128K	1-way	1.00 + O(1)	308.6	0.998
	16K to 128K	1- to 4-way	1.12 + O(1)	312.1	1.009
	16K	4-way	1.28 + O(1)	312.5	1.011
	1K to 8K	1- to 4-way	1.47 + O(1)	327.2	1.058
All-associativity	16K	1-way	1.00 + O(1)	300.8	1.0
	16K	4-way	1.28 + O(1)	309.2	1.028
	16K to 128K	1-way	> 1.39 + O(1)	332.3	1.105
	16K to 128K	1-, 2- & 4-way	> 1.39 + O(1)	366.6	1.219
	1K to 8K	1-way	> 2.65 + O(1)	402.9	1.339

Table 2-2. MVS Run-time vs. Simulation.

This table shows the run-time predictions and simulation times of forest, stack and all-associativity simulations for various cache configurations with the *mvs* trace. The third and fifth stack simulations, not used in previous figures, use 512 and 32 stacks, respectively.

Run-time predictions correctly order simulation runs of each algorithm, but not simulation runs of different algorithms. A least squares fit shows that the simulation time per reference is dominated by the constant term. The fit for forest simulation is $31.7*x + 279.7$ seconds/1M-references where x is the first term of the run-time prediction. The fit for stack simulation is $34.6*x + 273.2$ seconds/1M-references. Consequently a simulation with run-time $1.10 + O(1)$ is likely to be one percent slower than a simulation with run-time $1.00 + O(1)$.

evaluating multiple direct-mapped caches.

Since I find inclusion usually does not hold for alternative set-associative CPU cache designs, I describe an algorithm for simulating alternative set-associative caches that does not use inclusion. This algorithm, called *all-associativity simulation*, can be made faster be taking advantage of *set hierarchy*, a necessary, but not sufficient condition for inclusion, since I find set hierarchy usually holds between set-associative caches (e.g., those that use bit selection). With set hierarchy, the time to run most all-associativity simulations is within 30 percent of the time of one stack or forest simulation. Thus while forest simulation enables the rapid simulation of direct-mapped caches, all-associativity simulation facilitates the rapid simulation of direct-mapped and set-associative caches.

The principal impact of this work is that all-associativity simulation with set hierarchy allows a similar or wider cache design space to be examined in comparable or less simulation time than required with stack simulation. As shown in Figure 2-23, six stack simulations are needed to simulate mixed caches of one block size with associativities of one-, two-, and four-way, and sizes 16K, 32K , 64K and 128K-bytes. With six all-associativity simulations, requiring comparable time for practical CPU caches†, one can evaluate mixed, instruction-only and data-only caches of two block sizes and numerous associativities and sizes. In particular the use of all-associativity simulation facilitated the evaluation of the large number of alternative CPU cache designs I consider in the next chapter.

† E.g., CPU caches that are less than or equal to 32-way set-associative and use bit selection to map references to sets.

2.8. References

[Agar86] A. Agarwal, R. L. Sites and M. Horowitz, ATUM: A New Technique for Capturing Address Traces Using Microcode, *Proc. Thirteenth International Symposium on Computer Architecture* (June 1986).

[Bela66] L. A. Belady and J. Gecsei, A Study of Replacement Algorithms for a Virtual-Storage Computer, *IBM Systems Journal*, 5, 2 (1966).

[Bell74] J. Bell, D. Casasent and C. G. Bell, An Investigation of Alternative Cache Organizations, *IEEE Trans. on Computers*, C-23, 4 (April 1974), 346-351.

[Benn75] B. T. Bennett and V. J. Kruskal, LRU Stack Processing, *IBM Journal of R & D* (July 1975).

[Clar83] D. W. Clark, Cache Performance in the VAX-11/780, *ACM Trans. on Computer Systems*, 1, 1 (February, 1983), 24 - 37.

[Digi81] Digital Equipment Corp., VAX Architecture Handbook (1981).

[Good83] J. R. Goodman, Using Cache Memory to Reduce Processor-Memory Traffic, *Proc. Tenth International Symposium on Computer Architecture*, Stockholm, Sweden (June 1983), 124-131.

[Haik84] I. J. Haikala and P. H. Kutvonen, Split Cache Organizations, CS Report C-1984-40., Univ. of Helsinki (August 1984).

[Hill84] M. D. Hill and A. J. Smith, Experimental Evaluation of On-Chip Microprocessor Cache Memories, *Proc. Eleventh International Symposium on Computer Architecture*, Ann Arbor, MI (June 1984).

[Hill85] M. D. Hill, DineroIII Documentation, Unpublished Unix-style Man Page, University of California, Berkeley (October 1985).

[Matt70] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger, Evaluation techniques for storage hierarchies, *IBM Systems Journal*, 9, 2 (1970), 78 - 117.

[Olke81] F. Olken, Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies, Masters Report, Lawrence Berkeley Laboratory LBL-12370, University of California, Berkeley (May 1981).

[Patt83] D. A. Patterson, P. Garrison, M. D. Hill, D. Lioupis, C. Nyberg, T. N. Sippel and K. S. V. Dyke, Architecture of a VLSI Instruction Cache for a RISC, *Proc. Tenth International Symposium on Computer Architecture*, Stockholm, Sweden (June 1983), 108-116.

[Puza85] T. R. Puzak, Analysis of Cache Replacement Algorithms, Unpublished Ph.D. Dissertation, Dept. of Electrical and Computer Engineering, University of Massachusetts (February 1985).

[Smit78] A. J. Smith, A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory, *IEEE Trans. on Software Engineering*, SE-4, 2 (March 1978), 121-130.

[Smit85] A. J. Smith, Cache Evaluation and the Impact of Workload Choice, *Proc. Twelfth International Symposium on Computer Architecture* (June 1985).

[Stre76] W. D. Strecker, Cache Memories for PDP-11 Family Computers, *Proc. Third International Symposium on Computer Architecture* (January 1976), 155-158.

[Thom86] J. G. Thompson, A. J. Smith and F. Olken, *Efficient Trace-Driven Analysis Techniques for Memory Hierarchies*, University of California, Berkeley, (June 1986). Draft.

[Thom87] J. G. Thompson, Efficient Analysis Of Caching Systems, Computer Science Division Technical Report UCB/Computer Science Dpt. 87/374, University of California, Berkeley (October 1987).

[Trai71] I. L. Traiger and D. R. Slutz, One-Pass Techniques for the Evaluation of Memory Hierarchies, IBM Technical Report RJ 892 (#15563) (July, 1971).

3 The Effect of Set-Associativity on Cache Memory Performance

3.1. Introduction

This chapter uses the algorithms developed in Chapter 2 to study direct-mapped and set-associative caches. While there have been a considerable number of papers on cache analysis, few concentrate on the effects of changing associativity or discuss implementation considerations. I do both in order to show that large direct-mapped caches often have smaller effective access times than set-associative caches, despite having larger miss ratios. For this reason I argue that large, simple caches are superior to more complex cache designs when caches become sufficiently large.

I use *miss ratio* and *effective access time* to measure the performance of caches that use LRU replacement and do no prefetching. Analysis with miss ratio shows that the miss ratio of a direct-mapped cache is roughly proportional to the miss ratio of a two-way set-associative cache of the same size. This result implies that the absolute miss ratio difference between the two cache designs will decrease as both caches get larger. In particular, the data show that decreasing the degree of associativity from two to one (direct-mapped) increases the miss ratio by 20 to 40 percent relative to the two-way set-associative cache.

Analysis with effective access time combines cache implementation details and the above miss ratio analysis. An examination of cache implementations shows that direct-mapped caches have faster cache access (hit) times than do set-associative caches, principally because direct-mapped caches do not have to multiplex together multiple candidate words. As cache sizes increase, the access time difference becomes more important than the miss ratio difference, causing the effective access time of a direct-mapped cache to improve relative to that of a set-associative cache of the same size. I find, for example, that the effective access times of direct-mapped caches are similar (within five percent) or slightly better than those of set-associative caches for cache sizes of 32K bytes and larger.

Before analyzing caches with miss ratio and effective access time, I discuss cache performance metrics, my methods and previous work.

3.1.1. Cache Performance Metrics

The most commonly-used cache performance metric is *miss ratio* [Smit82]. The miss ratio for a cache C is†:

$$m(C) = \frac{\text{the number of misses with cache } C}{\text{the number of processor memory references}}$$

Miss ratio is used because it is easy to define, interpret, compute [Matt70], and perhaps most important, is implementation-independent. This independence facilitates cache performance comparisons between caches not yet implemented and those implemented with different technologies and in different kinds of systems. Unfortunately some comparisons of dissimilar caches with miss ratio can lead to misleading results. A miss ratio comparison, for example, between the Cray-1 instruction buffers [Siew82] and the Motorola 68020 on-chip instruction cache [MacG85] is meaningless because the technologies and workloads have little in common.

Since miss ratio comparisons contrast the number of misses, they can be misleading if the penalty for a miss varies. For instance, increasing cache block size often reduces the number of misses and hence the miss ratio, but it often also increases the number of cycles needed to load a block. The actual change in cache performance will depend on how much the number of misses decreases and on how much the time to service a miss increases [Smit87].

Another commonly-used cache performance metric is *effective access time*, $t_{eff}(C)$. Effective access time is the average latency, as seen by the processor, required by the memory system to service a memory reference. In this chapter, I model it as:

$$t_{eff}(C) = t_{cache}(C) + m(C)*t_{memory}(C)$$

where $m(C)$, $t_{cache}(C)$ and $t_{memory}(C)$ are the miss ratio, access time and average miss penalty for cache C. Many cache memory analyses with effective access time assume that $t_{cache}(C)$ is the same for all caches studied. I do not make this assumption in my analysis, because I expect cache access time differences to be important in the analysis of large caches where miss ratios and miss ratio differences are small. The results of Section 3.3.3, indeed, show that ignoring access time differences when comparing large caches of varying associativities can lead to incorrect conclusions.

The principal advantage of using effective access time over using miss ratio is that using effective access time allows caches with different hit and miss times to be more accurately compared. One can, for example, determine whether increasing cache block size improves performance. The principal disadvantage of effective access time is that implementation details must be examined and assumptions must be made for the values of $t_{cache}(C)$ and $t_{memory}(C)$. Performance estimates with any implementation assumptions are less general, and those with incorrect assumptions are misleading.

Caches can also be compared with system performance metrics, such as *benchmark execution time* or *effective number of processors*. While these metrics are useful for comparing similar alternative caches within the context of an existing system, they seldom produce conclusions that generalize to other designs, because of the difficulty of isolating cache effects from other system effects. For this reason I compare caches with miss ratio and effective access time, but not with system performance metrics.

† I use $m(C)$, rather than m, to emphasize that the miss ratio is a function of a cache organization. C represents all attributes of a cache. I will use $C(A=1)$ and $C(A=2)$ to denote two similar caches with different associativity.

3.1.2. Methods

I analyze caches with trace-driven simulation for the same reasons as Smith [Smit82, Smit85]. The principal advantage of trace-driven simulation is that numerous caches parameters can be varied across of a wide range of values in repeatable experiments, while its principal disadvantage is that workload samples must be relatively short, due to disk space and simulation time limits. The traces I use, containing a total of tens of millions of references, represent only tens of CPU seconds on a VAX-11/780-class machine. The algorithms developed in Chapter 2 mitigate problems with simulation time limits by reducing total simulation time through simulating 50 cache configurations concurrently almost as fast as was required to simulate an individual cache.

A second disadvantage of trace-driven simulation has been the lack of the traces that include operating system and multiprogramming effects. However, I have been able to obtain traces that include these effects (see Tables 3-1, 3-2 and 3-3). Consequently, I use these traces directly rather than trying to model operating system references and multiprogramming effects.

Unfortunately the most of traces in Table 3-1 are made of several 500K-reference samples. To calculate overall miss ratios I derive steady-state miss ratios from short traces, which may not fill up large caches, and average the results from individual traces. Two estimates of the steady-state miss ratio are the *cold-start* and *warm-start* miss ratios [East75, Stre83]. The cold-start miss ratio, calculated by counting misses from an initially empty cache, gives an upper bound on the expected steady-state miss ratio. The warm-start miss ratio, calculated after a cache is *warm*, has an expected value closer to the steady-state miss ratio, but is based on less data since the beginning of each trace is ignored. A cache is warm if its future miss ratio is not significantly affected by the cache recently being empty [Agar87a]. Consequently, a full cache is clearly warm. Few trace references are discarded waiting for a small cache to become warm, since a small cache fills rapidly. Large caches, however, can require millions of references to fill up.

To save simulation time and enable the use of short trace samples, Agarwal et al. have defined a new method of determining when a cache is warm [Agar87a]. They plot cumulative *cold misses* versus cumulative memory references, where a *cold miss* is a miss that fills an empty cache block frame. They observe that most cumulative-cold-miss lines rise rapidly, hit a knee, and then rise slowly. They define a cache to be warm after the knee is encountered, since beyond this point the effect on the miss ratio of the cache being recently empty is small. Figure 3-1 shows an example of this method. The number of cold misses in the 128K-byte cache during the second half of the trace is about 200. Thus, residual cold-start effects cannot affect the miss ratio by more than a negligible 0.0008 (200/250K). I analyzed my traces, described in Table 3-1, and found that most caches are *warm* at or before 250K references. For this reason, I will compute steady-state miss ratios using the second 250K references of each 500K-reference trace sample.

The second problem in dealing with numerous traces is that of combining results to give overall numbers. I approach summarizing miss ratios by using the arithmetic mean of the miss ratios from reasonable trace samples. I omit from the mean traces with very low miss ratios (e.g., user-only traces) and traces from non-32-bit architectures. The arithmetic mean is reasonable for this task, because it represents the miss ratio of a workload consisting of an equal number of references from each of the traces.

I also examine the *ratio of miss ratios* between caches of one associativity and twice that associativity. Given a collection of traces a ratio of miss ratios can be calculated two ways, as the *ratio of averages* or as the *average of ratios*. Let $m_i(A=j)$ be the miss ratio for a cache of associativity j with trace i. The ratio of miss ratios between a direct-mapped and two-way set-associative cache, for example, is AVERAGE$[m_i(A=1)]$ / AVERAGE$[m_i(A=2)]$ calculated as the *ratio of averages* and AVERAGE$[m_i(A=1)/m_i(A=2)]$ calculated as the *average of ratios*, where AVERAGE computes the mean of it arguments. The principal advantage of average of the ratios, the latter method, is that it allows traces with unrealistically low miss ratios, but reasonable *ratios* of miss ratio to be included. Smith uses the average of ratios to examine caches of varying block size [Smit87]. One disadvantage of the average of the ratios, however, is that ratios with small denominators can have a large influence on

Trace Name	Architecture; Operating System	Comments
fortran	IBM 370; (FortG compiler)	A collection of samples from four fortran user traces: a factor analysis, analysis of satellite information, a numerical analysis and an FFT [Smit82].
mul2	VAX-11; VMS	A collection of four samples of a user and system trace of spice, a circuit simulator, and alloc, a microcode address allocator, running concurrently. This trace of multiprogrammed operation was gathered with microcode modifications on a VAX 8200 [Agar86].
mul8	VAX-11; VMS	A collection of three samples of a user and system trace of an eight-job multiprogrammed workload gathered with microcode modifications [Agar86]. The programs being run under VMS are spice, alloc, a Fortran compile, a Pascal compile, an assembler, a string search in a file, jacobi (a numerical benchmark) and an octal dump [Agar86].
ue	VAX-11; Ultrix	A collection of three user and system traces of synthetic multiprogrammed workloads with 2, 10 and 20 identical tasks [Agar86]. The task is a program called utep, which simulates an interactive user.
mvs	IBM 370; MVS	A collection of system calls from two Amdahl standard MVS workloads [Smit85].
synapse_devel	Synapse N+1 (68000); Synthesis	User and system trace of a synthetic development workload consisting of editing, compiling, linking and executing a collection of programs, as well as executing common system utilities (directory list, etc).
2nd500k	VAX-11 & IBM 370; VMS & MVS	A trace whose miss ratios are the arithmetic average of the miss ratios from second 500,000 references of mul2, mul8, ue, mvs1 and mv2. Mvs1 and mvs2 are the two parts of the mvs trace. These five traces include tracing of system code with context switching. The first three also include user code; the latter two do not.
atum	VAX-11; VMS & MVS	A collection of 23 of 33 trace samples gathered from ATUM and distributed by Dick Sites of DEC [Agar86]. It includes samples that are part of *mul2*, *mul8* and *ue*, and other samples that have lower absolute miss ratios (often around 0.0050), but reasonable ratio of miss ratios.

Table 3-1. Address Traces Used.

This table describes the traces used in these studies. References [Agar86, Smit82, Smit85].

the average ratio. I find empirically in Section 3.2.3 that the two methods produce comparable results, and therefore use the more convenient method, the ratio of averages, for most ratios of miss ratios I calculate.

3.1.3. Previous Work

Many papers have discussed associativity as part of a more general analysis; these include: [Lipt68], [Matt71], [Kapl73], [Bell74], [Stre76], [Smit82], [Haik84], [Alex86] and [Agar87a]. Most concentrate on caches smaller than 32K bytes. My associativity results are consistent with basic associativity results in these papers. In addition I extend analysis to larger caches. I discuss several of the more-recent of these papers below; technological changes have reduced the usefulness the earlier

Trace Name	Instruction References (%)	Length (1000's of references)	Size (K-bytes)
fortran	56	1213	80
	59	1213	65
	56	1213	36
	54	882	18
mul2	56	372	94
	52	386	204
	53	383	169
	56	367	165
mul8	51	408	218
	54	390	196
	46	429	194
ue	56	358	205
	57	372	191
	55	364	221
mvs	52	1000	200
	53	1000	340
synapse_devel	62	1048	1378
	68	1048	124
	68	1048	100
	68	1048	105
mul2_2nd500K	53	500	218
mul8_2nd500K	51	500	292
ue_2nd500K	55	500	277
mvs1_2nd500K	52	500	163
mvs2_2nd500K	55	500	201

Table 3-2. Data on Traces.

This table presents data on the address traces used. The first column gives the name of each trace. The second gives the fraction of all references that are instruction references. In these simulations I do not distinguish between data reads and writes. The third column gives the length of the address traces in 1000's of references. The final columns gives the number of distinct bytes referenced by the trace, where any reference in an aligned 32-byte block touches the whole block. The size is computed by multiplying the number of 32-byte blocks touched by 32.

papers.

While many papers discuss associativity, only one by Smith [Smit78] concentrates exclusively on associativity. In this paper, Smith proposes a model which effectively explains most of the miss ratio differences between caches of varying associativity, and then he validates this model for some set-associative caches. In Section 3.2.2, I explain the model and show that it applies to a wider range of caches, including large direct-mapped caches.

Several recent papers present considerable data on caches, in general, which needs to be considered when designing cache studies. The most comprehensive survey of cache design and performance to date is by Smith [Smit82]. Smith surveys aspects of cache design, including cache prefetch algorithms, associativity, block size, replacement algorithms, write-through vs. copy-back, approximating multiprogramming, I/O through the cache, and splitting the cache by data vs. instructions or user vs. supervisor. He also justifies the trace-driven simulation and uses it develop numerical results for all of the cache aspects surveyed. He presents results, based primarily on user-only traces, for caches as large as 64K bytes. Large caches (≥ 32K bytes) with restricted associativity (two-way set-associative or direct-mapped), however, are not studied in order to reduce simulation time; for example, his default

Trace Name	Instruction References (%)	Length (1000's of references)	Size (K-bytes)
dec0	50	362	120
	50	353	125
fora	52	388	144
forf	52	401	128
	53	387	152
	53	414	105
	52	368	205
fsxzz	51	239	104
ivex	60	342	210
macr	55	343	199
memxx	49	445	139
mul2	52	386	204
	53	383	169
	56	367	165
mul8	51	408	218
	54	390	196
	46	429	194
null	58	170	55
savec	50	432	94
	61	228	54
ue	56	358	205
	57	372	191
	55	364	221

Table 3-3. Data on More Traces.

Like Table 3-2, this table presents data on the address traces used. I refer to this collection of traces as *atum*, since all were gathered with ATUM [Agar86]. Data for traces *mul2*, *mul8* and *ue* is repeated here, since these trace samples are used separately and together with the rest of the *atum* traces.

64K-byte cache is 32-way set-associative.

Smith extended his survey of caches in 1985 by examining the impact of trace selection on miss ratio results [Smit85]. He warns cache designers that miss ratio predictions from small, user-only traces are usually optimistic, and presents some *design target miss ratios* for mixed (unified), instruction and data caches with 16-byte blocks and fully-associative placement to serve as "rules of thumb" for cache designers working with "a 32-bit architecture running fairly large programs and a mature (i.e., large) operating system." Smith derives the design target miss ratios from his own opinions, based on the data in the paper, his other direct experience with cache design and his knowledge of the cache literature. I find these design target miss ratios consistent with my results, and I extend them to caches of varying associativity.

In another paper Smith concentrates on cache block size [Smit87]. He isolates the effects of block size from other factors by using as his principal metric the ratio of the miss ratio of a cache with one block size to the miss ratio of a cache with twice that block size. He uses these ratios to extend his design target miss ratios to caches with block sizes from 4 to 128 bytes, and combines these miss ratios with some cache access and miss times to show that the effective access times of caches connected to high-performance microprocessors are minimized for blocks size between 16 and 64 bytes, and that larger blocks are suitable for mainframes with higher memory bandwidth.

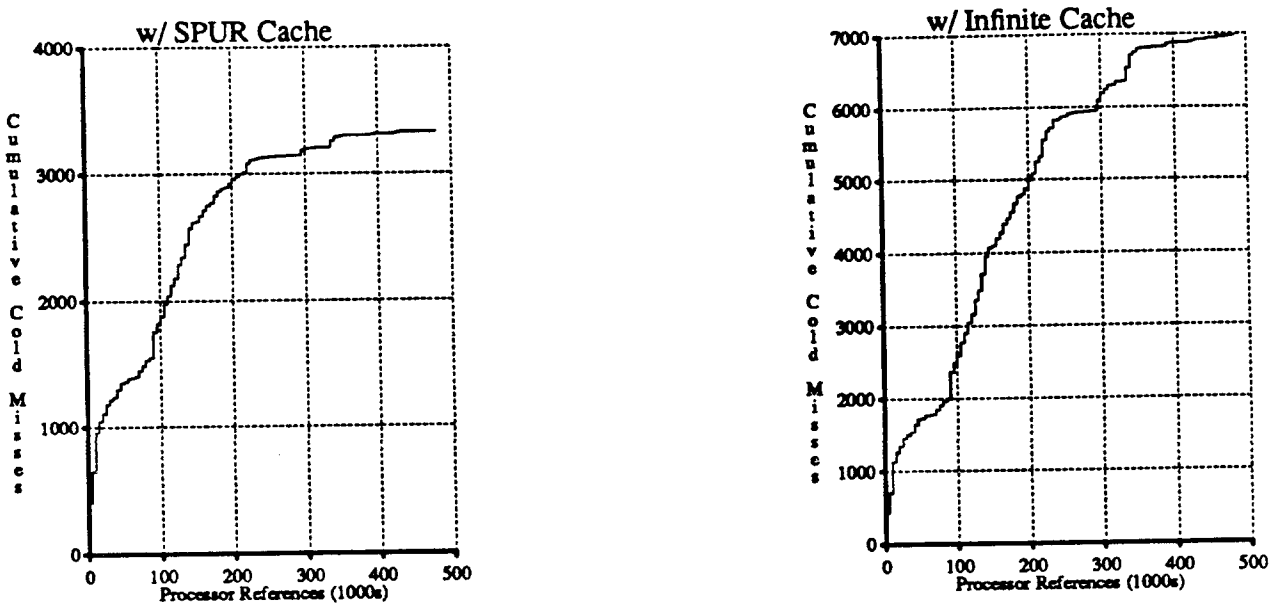


Figure 3-1. Cumulative Cold Misses for "mul2".

This figure shows cumulative *cold* misses versus processor references for two cache sizes with trace *mul2*. A *cold* miss fills a previously empty cache block frame. A cache is full if a cold miss has occurred at each block frame; a cache is warm if the rate of cold misses is small relative to the overall miss ratio.

The left plot shows that 3129 of the 4096 cold misses in the SPUR cache (128K-bytes, direct-mapped, 32-byte blocks) occur by 250K references, and the subsequent rate of cold miss is small (only 198 in the next 250K references). Therefore, for this trace, a 128K cache is warm after 250K references.

The right plot shows misses in an infinite cache, for which all misses are cold misses. An infinite cache is a cache so large that it never replaces any blocks. The slower rate of misses in the second half of the trace is due to the cache getting warm or a variation in input trace behavior. Since other traces exhibited a slower rate of cold misses after 250K references, I conclude that even an infinite cache is getting warm after 250K references.

The study of large caches requires the consideration of operating system and multiprogramming effects. Clark takes these effects into account by directly monitoring an operating computer, a VAX-11/780 [Clar83]. His method allows long workload samples to be used, but constrains the caches measured to be the 780's cache (8K bytes, two-way set-associative, 8-byte blocks) or half of the 780's cache (4K bytes, direct-mapped, 8-byte blocks). For this reason Clark can say little about large caches. The paper is, however, an excellent, detailed study of the 780's cache.

Haikala and Kutvonen present trace-driven simulation results, based six traces of compilers running on the stack-oriented Burroughs B7800 [Haik84]. Part of this paper examines associativity, finding results consistent with my miss ratio results. Without specifically examining cache implementations, the authors also claim that two-way set-associativity is best for most caches, given the marginal performance improvement and greater implementation complexity of higher associativity. My results agree with this conclusion for the small cache sizes ($\leq$ 16K bytes), but find direct-mapped caches offer similar or better performance at large cache sizes, for implementation reasons that Haikala and Kutvonen do not consider.

Alexander et al. use trace-driven simulation to study small and large caches [Alex86]. They gather two 3 million-reference traces by using a hardware monitor to read 4K addresses directly off the backplane of a SYS32 computer, holding its real-time clock and uploading the 4K addresses to another computer for subsequent simulation, releasing the clock, and repeating. The SYS32 computer uses a National 32016 microprocessor, runs a derivative of 4.1 BSD Unix, and has 1M byte of main memory. Even though Alexander et al. collapse consecutive 16-bit references into 32-bit references, their results are optimistic. While Clark finds miss ratios for the 780's cache of 10 to 17 percent, Alexander et al.

predict 5 to 6 percent. While Smith predicts a miss ratio of 4 percent for a 32K-byte cache (unified, fully-associative, 16-byte blocks) [Smit85], Alexander et al. predict 0.9 to 1.7 percent for a similar cache. Other data gathered with a hardware monitor also predicts a 3 to 4 percent miss ratio for the above cache [Smit82].

Agarwal et al. use traces gathered by modifying the microcode of the VAX 8200 to study large caches and try to separate operating system and multiprogramming effects [Agar87a]. They briefly examine associativity, where they find that associativity in large caches impacts multiprogramming workloads more strongly than uniprocessor workloads. They find for one workload that decreasing associativity from 2-way to direct-mapped increases the multiprogramming miss ratio by 100 percent and the uniprogramming miss ratio by 43 percent. While the quantitative changes are much larger than I observe, the more important qualitative conclusion they make is that large cache results are misleading unless operation system and multiprogramming effects are taken into account.

Finally, few cache studies have considered implementation details or analyzed caches with effective access time. Four papers that do are: (1) a paper by Kaplan and Winder [Kapl73], which stresses the value of assessing caches with effective access time and instructions per cycle; (2) Smith's paper on block size [Smit87] (described earlier in this section), where Smith uses his design target miss ratios and some implementation assumptions to determine optimal block sizes; (3) a paper by Agarwal et al. on designing an instruction cache on a single-chip RISC processor [Agar87b], which I describe in the next chapter on on-chip instruction memory; and (4) a paper by Chang, Chao and So, discussed below, which investigates a cache with a non-uniform access times [Chan87].

Chang, Chao and So examine designing a physically-tagged, single-cycle, 128K-byte cache for a System/370 CPU, implemented with several large CMOS chips on a multi-layer ceramic module. To achieve a single-cycle access time without lengthening the cycle time too much, they found it necessary to access the cache and translation buffer in parallel. A straight-forward implementation of such a 128K-byte cache requires it to be 32-way set-associative so that it can be indexed with address bits from within the page offset (that are not changed by address translation). Estimates showed, however, that the access time of this cache, "which involved two chip-crossings, a cache directory look-up, a comparator and several multiplexors, would be about 30% longer than the targeted CPU cycle time" [Chan87]. To reduce average access time, Chang, Chao and So considered building a 32-way set-associative cache with a one-cycle access to the most-recently-used (MRU) block of each set and a two-cycle access to all other cache blocks. The collection of blocks accessed in one cycle, called the *MRU region*, can be thought of as a direct-mapped 4K-byte cache within a cache. Additional analysis revealed, however, that performance could be further improved by *reducing* associativity from 32-way to four-way, thereby increasing MRU size to 32K bytes, but requiring two more index bits. Since these index bits are not in the page offset, they are not available until after address translation. This problem did not impact the access time of Chang, Chao and So's cache, because they were able to use these bits to control a four-to-one multiplexor at the end of the cache lookup.

While Chang, Chao and So give convincing evidence that an MRU cache is superior to set-associative caches, they present no evidence on how it compares with a large direct-mapped cache. In Section 3.3.4 I examine MRU-cache designs with respect to direct-mapped caches when address translation does not interact with a cache lookup (e.g., when a virtually-tagged cache is used). I find that the advantage of the MRU scheme is modest, and that the advantage can be easily overwhelmed by implementation disadvantages.

3.2. Analysis with Miss Ratio

In this section, I examine direct-mapped and set-associative caches with miss ratio results from trace-driven simulation. I first present the raw miss ratios and suggest a way to divide a miss ratio into three components to bolster one's intuition regarding which cache design changes can reduce the miss ratio. Then I use a method proposed by Smith in 1978 for predicting set-associative miss ratios from LRU distance probabilities [Smit78]. I show that this method accurately predicts the miss ratios of direct-mapped and set-associative caches from the miss ratios of all fully-associative caches. Finally, I

examine the relative miss ratio increase caused by reducing associativity, which I call *miss ratio spread*, and use it to extend Smith's design target miss ratios to caches of varying associativity.

3.2.1. Raw Miss Ratios

In the course of this research I have simulated over 150 cache configurations with traces of over 30 different programs, running alone or in a multiprogrammed workloads. The traces used are described in Tables 3-1, 3-2 and 3-3. Initially, I examine mixed (i.e., cache data and instructions together) caches that do not prefetch blocks, use LRU replacement, and have a 32-byte blocks. I vary associativity (1- to 8-way set-associative and fully-associative) and cache size (256, 512, ..., 256K and infinite). An infinite cache never replaces a block so its miss ratio is a lower bound on the miss ratio with a given block size. Later I also vary cache type (to instruction only and data only) and block size (to 16 and 64 bytes).

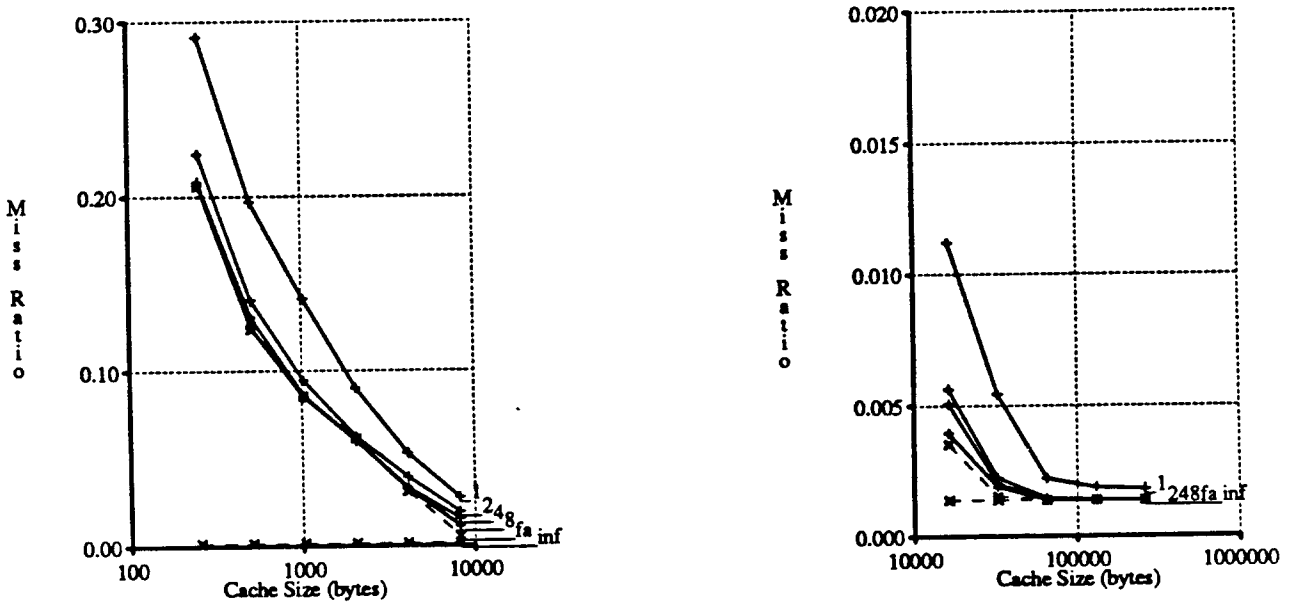


Figure 3-2. Cold-start Miss Ratios for Some User Traces.

This figure shows cold-start miss ratios for various caches. Each miss ratio is the arithmetic average of the miss ratios from four System/360 FORTRAN user traces (see Table 3-2). Each cache is mixed (i.e., cache data and instructions together), does not prefetch blocks, uses LRU replacement, has a 32-byte block size and is simulated from a cold-start. Cache associativity and size are varied. The associativities studied are one-way set-associative (direct-mapped, labeled '1'), two-way set-associative ('2'), four-way set-associative ('4'), eight-way set-associative ('8') and fully-associative (a dashed line, 'fa'). The cache sizes examined are 256 to 256K bytes and infinite (a dashed line,'inf'). An infinite cache, which never replaces an active block, has the lowest miss ratio possible for a given block size (so long as prefetching is not permitted).

While useful for examining small caches (≤ 8K bytes), these user-only traces provide little information about larger caches, since their large cache miss ratios rapidly approach the infinite cache miss ratio.

The results with single-process, user-only traces are illustrated by using four FORTRAN traces (see Figure 3-2). Results for small caches, ≤ 8K bytes, confirm the data of others that as cache size or associativity is increased, miss ratios get smaller, but at a decreasing rate [Alex86, Smit82, Smit85]. The miss ratios for large caches, especially those greater than 32K bytes, are virtually identical and near the infinite cache miss ratio of 0.0014, which implies that all large caches perform near the infinite cache miss ratio. Results for other single-process, user-only traces, including VAX Unix traces [Smit85] and MIPS traces are not shown, because they also support the conclusion that increasing cache size beyond 64K bytes is not worthwhile. To see that this conclusion is wrong, I next examine caches with traces where the effects of operating system references and context switching are included.

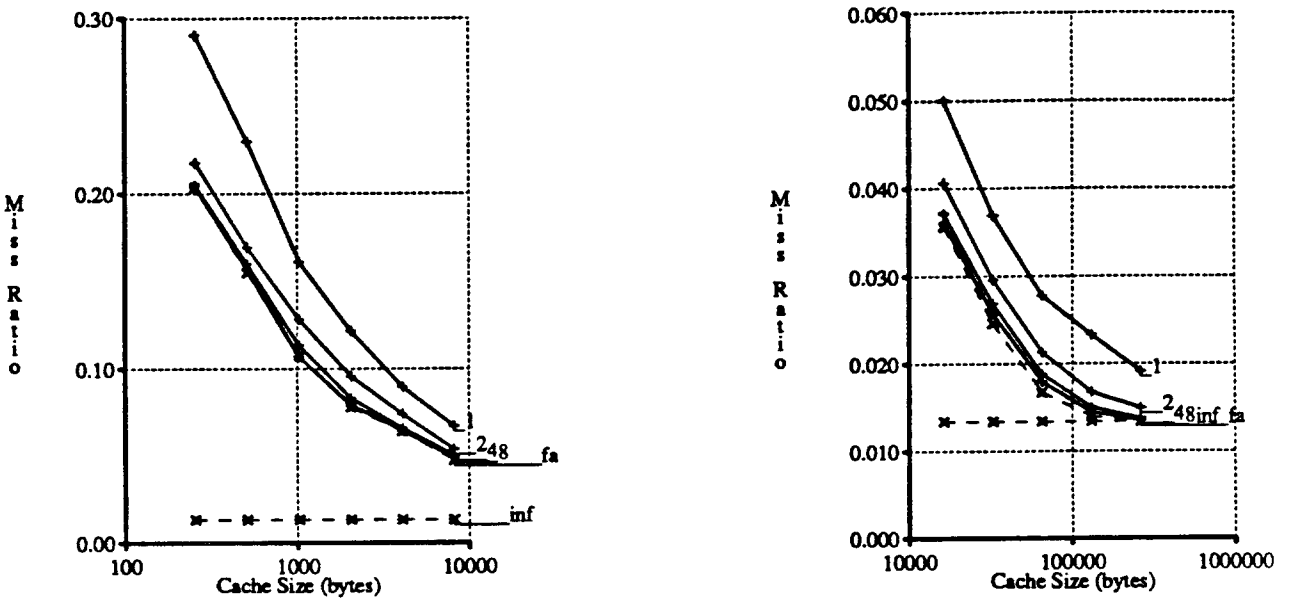


Figure 3-3. Cold-start Miss Ratios for "mul2".

This figure shows cold-start miss ratios for various caches, using an average of three samples from the "mul2" VAX-11 user/system trace. It examines the same cache configurations as Figure 3-2.

Results for large caches are more believable than for the FORTRAN user-only traces since most miss ratios are larger than the infinite cache miss ratio.

Figure 3-3 portrays typical results with system references and multiprogramming effects included. Figure 3-3 shows that as cache size or associativity is increased, miss ratios get smaller, but at a decreasing rate, for caches up to 256K bytes. Figures 3-4 and 3-5 illustrate results for other workloads.

Miss ratios from individual traces vary widely. A greater quantitative understanding of the data can be facilitated by examining average miss ratios, if one remembers that the underlying data are variable. I have computed average miss ratios with the miss ratios from the second 500,000 references of traces *mul2, mul8, ue, mvs1* and *mvs2*. For brevity, I refer such an average miss ratio as the miss ratio for *trace 2nd500k*. Each of the traces is from a 32-bit architecture, includes operating system references, and has non-trivial miss ratios with large caches. Figure 3-6 shows miss ratios for mixed caches with 32-byte blocks.

The miss ratios in Figure 3-6 and the other miss ratios presented so far are pessimistic relative to steady-state miss ratios, because they are cold-start, not warm-start miss ratios. To obtain more accurate estimates for steady-state miss ratios, I calculate warm-start miss ratios by not counting misses and references in the first 250K references of each trace. This approach is justified in Section 3.1.2. Figure 3-7 and Tables 3-4, 3-5, and 3-6 display warm-start miss ratios.

The miss ratios for trace *2nd500k* are consistent with other data from other sources, as illustrated in Figure 3-8 [Agar87a, Smit82, Smit85, Smit87]. At larger cache sizes, some miss ratios are more pessimistic than Agarwal et al.'s. These miss ratios are also consistent with 10 to 17 percent miss ratio Clark found for the VAX-11/780 cache (two-way set-associative 8K-byte cache with 8-byte blocks) by monitoring the hardware [Clar83]. I report a 8 to 9 percent miss ratio for a two-way set-associative 8K-byte cache with *16-byte* blocks. Since Smith expects an 8-byte miss ratio to be 72 percent larger than the 16-byte miss ratio [Smit87], I predict a reasonable 14 to 16 percent miss ratio for the 780 cache.

When examining miss ratio data for numerous cache organizations, I have found it useful to partition misses, or miss ratios, into three components intuitively based on the cause of the misses, but calculated from the miss ratios of related caches. For some cache C:

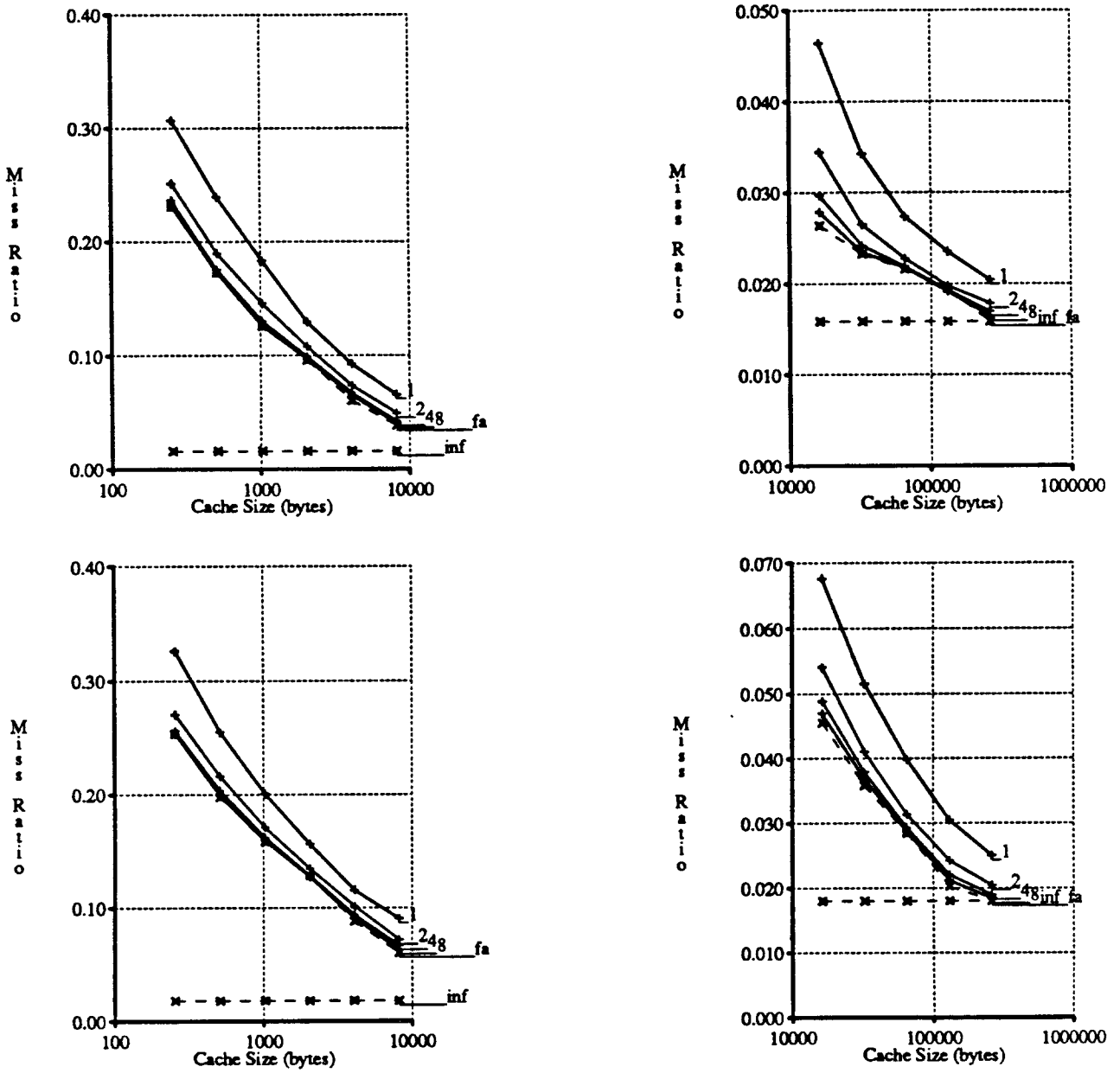


Figure 3-4. Cold-start Miss Ratios.

For "mul8" (top) and "ue" (bottom).

(1) The *compulsory* component of the miss ratio arises from misses to previously-unreferenced blocks. This component is equal to the miss ratio of an infinite cache with the same block size as cache C. Compulsory misses cannot be avoided with demand fetching and a fixed block size.

(2) The *capacity* component of the miss ratio arises from misses due to the finite size of cache C. It equals the miss ratio of a fully-associative cache similar to cache C (i.e, same size, block size and replacement algorithm), less cache C's compulsory miss ratio. Capacity misses cannot (usually) be avoided by caches of a given size and replacement algorithm, because full-associative placement usually yields a smaller miss ratio than set associative placement.

(3) The *(set-)conflict* component of the miss ratio arises from misses due to too many active blocks mapping to some sets. It is equal to the miss ratio of cache C, less the sum of the compulsory and capacity miss ratios. While both the compulsory and capacity miss ratio must be non-negative,

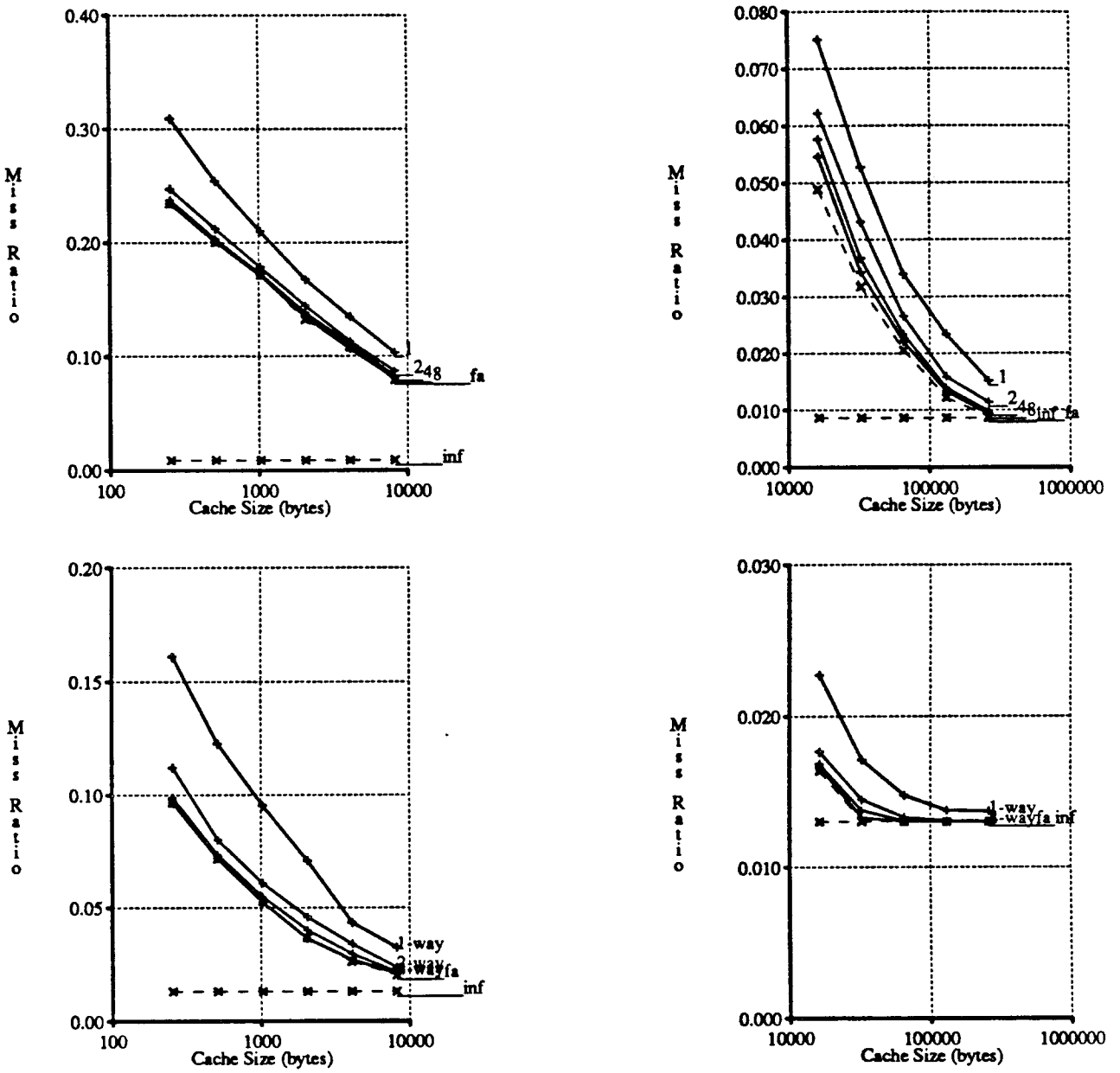


Figure 3-5. More Cold-start Miss Ratios.

For "mvs" (top) and "synapse_devel" (bottom).

the conflict miss ratio can be less than zero. In practice, negative conflict miss ratios, meaning a set-associative cache has a lower miss ratio than a fully-associative one, occur only for small instruction caches where locality is manifested predominantly as looping behavior [Smit83].

While dividing the miss ratios in these three components does not allow other cache miss ratios to be predicted, it does provide some insight into how the miss ratios of similar caches relate, and it allows the following negative statements to be made for caches of fixed block size: (1) If the conflict miss ratio is small, increasing the associativity cannot significantly improve the miss ratio. (2) If the conflict and capacity miss ratios are small, nothing can be done to improve the miss ratio. (3) Nothing can be done to improve miss ratios below the compulsory miss ratio.

Table 3-7 illustrates this method for trace "ue." I will examine compulsory, capacity and conflict misses with other traces in other papers. Results for this trace show that capacity misses dominate until

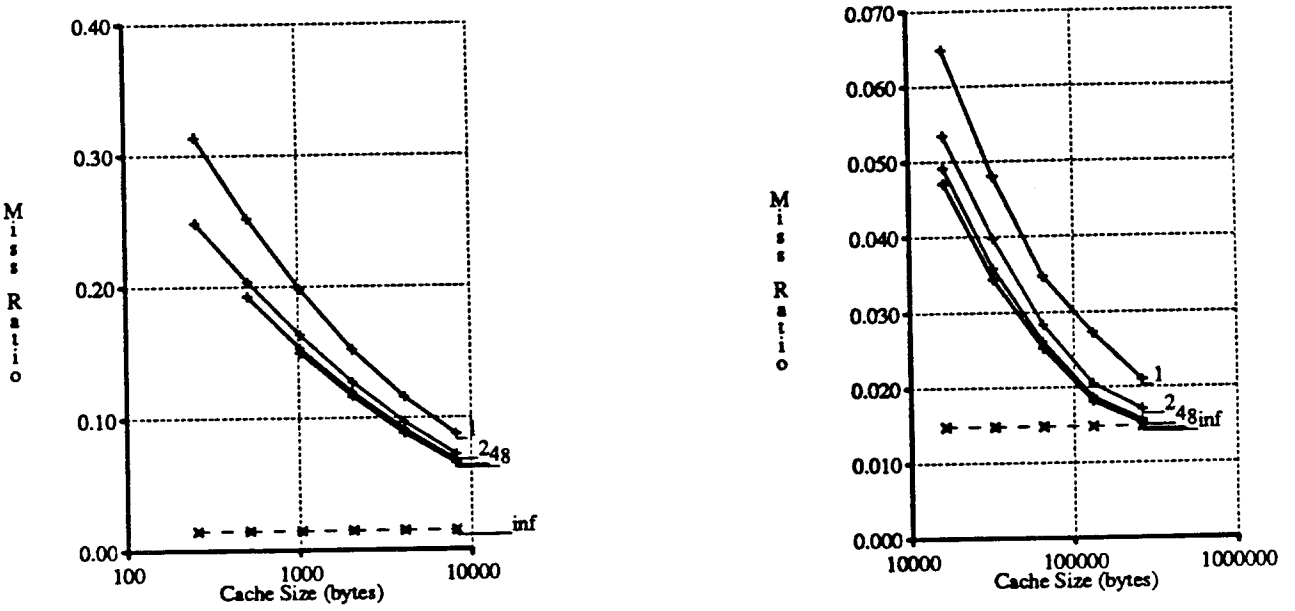


Figure 3-6. Cold-Start Miss Ratios for "2nd500k".

This figure shows the cold-start miss ratios with trace *2nd500k* for mixed caches with 32-byte blocks. Miss ratios for *2nd500k* are the arithmetic average of miss ratios of five different traces that include system and multiprogramming effects. The general shape of these average miss ratio lines are consistent with the shape for the individual traces. Consequently, averaging the individual miss ratios is not misleading.

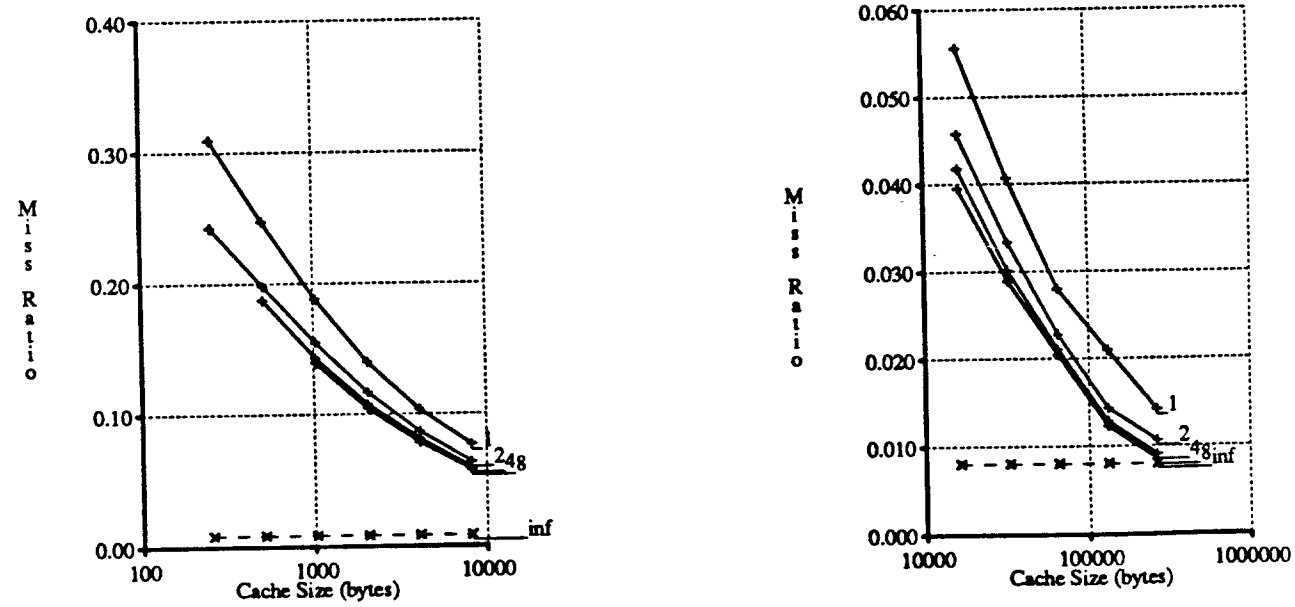


Figure 3-7. Warm-start Miss Ratios for "2nd500k".

This figure shows the warm-start miss ratios with trace *2nd500k* for mixed caches with 32-byte blocks. Caches are considered warm after 250K references. For these traces, warm-start miss ratios are 0.5 to 1.0 percent smaller (absolute) than cold-start miss ratios (see Figure 3-6). Consequently the relative decrease is only important for large caches.

cache size exceeds 64K bytes, at which time conflict misses dominate. Results for large caches, however, are unstable, since small miss ratio variations can introduce large relative changes in the size of miss ratio components. A large conflict miss ratio does not necessarily imply that associativity should be increased, since conflict misses can be reduced either by increasing associativity (set size) or by increasing cache size (the number of sets). Many conflict miss ratios for eight-way set-associative caches are near zero, since eight-way set-associative caches and fully-associative caches perform similarly. Little significance should be attached to the fact that some are negative by absolute amounts of 0.0006 and less.

Warm-Start Miss Ratios for Mixed Caches				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
256	1-way	0.3330	0.3093	0.3420
256	2-way	0.2864	0.2422	0.2406
512	1-way	0.2758	0.2470	0.2521
512	2-way	0.2386	0.1981	0.1852
512	4-way	0.2325	0.1872	0.1672
1024	1-way	0.2178	0.1882	0.1822
1024	2-way	0.1898	0.1550	0.1381
1024	4-way	0.1787	0.1432	0.1283
1024	8-way	0.1747	0.1393	0.1242
2048	1-way	0.1714	0.1398	0.1276
2048	2-way	0.1491	0.1169	0.1016
2048	4-way	0.1394	0.1081	0.0919
2048	8-way	0.1355	0.1044	0.0881
4096	1-way	0.1297	0.1033	0.0920
4096	2-way	0.1123	0.0866	0.0723
4096	4-way	0.1064	0.0815	0.0667
4096	8-way	0.1022	0.0786	0.0640
8192	1-way	0.0983	0.0767	0.0666
8192	2-way	0.0826	0.0636	0.0521
8192	4-way	0.0769	0.0591	0.0481
8192	8-way	0.0755	0.0576	0.0464
16384	1-way	0.0716	0.0556	0.0469
16384	2-way	0.0594	0.0457	0.0369
16384	4-way	0.0543	0.0418	0.0334
16384	8-way	0.0510	0.0395	0.0318
32768	1-way	0.0520	0.0407	0.0338
32768	2-way	0.0426	0.0333	0.0263
32768	4-way	0.0388	0.0302	0.0233
32768	8-way	0.0374	0.0289	0.0222
65536	1-way	0.0360	0.0280	0.0230
65536	2-way	0.0291	0.0228	0.0182
65536	4-way	0.0263	0.0211	0.0170
65536	8-way	0.0252	0.0205	0.0165
131072	1-way	0.0272	0.0210	0.0171
131072	2-way	0.0194	0.0144	0.0113
131072	4-way	0.0171	0.0129	0.0104
131072	8-way	0.0160	0.0123	0.0100
262144	1-way	0.0196	0.0142	0.0108
262144	2-way	0.0155	0.0107	0.0078
262144	4-way	0.0137	0.0091	0.0065
262144	8-way	0.0132	0.0085	0.0060

Table 3-4. Mixed Cache Miss Ratios.

This table shows *warm-start* miss ratios for mixed caches with LRU replacement and no prefetching on trace *2nd500k*, which is an average of user/system traces from the VAX-11 and system traces from the IBM 370. The miss ratios for 256-byte caches with associativity 4 and 8 and 512-byte caches with associativity 8 were not simulated to reduce simulation time. All caches are warmed-up for 250K references.

Warm-Start Miss Ratios for Instruction Caches				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes)		
		16	32	64
256	1-way	0.2343	0.1663	0.1265
256	2-way	0.2215	0.1511	0.1112
512	1-way	0.1966	0.1398	0.1080
512	2-way	0.1789	0.1219	0.0918
512	4-way	0.1739	0.1221	0.0940
1024	1-way	0.1545	0.1051	0.0766
1024	2-way	0.1432	0.0958	0.0690
1024	4-way	0.1399	0.0935	0.0671
1024	8-way	0.1391	0.0931	0.0675
2048	1-way	0.1224	0.0833	0.0607
2048	2-way	0.1156	0.0769	0.0543
2048	4-way	0.1107	0.0737	0.0525
2048	8-way	0.1099	0.0735	0.0526
4096	1-way	0.0938	0.0633	0.0461
4096	2-way	0.0857	0.0582	0.0411
4096	4-way	0.0818	0.0557	0.0396
4096	8-way	0.0791	0.0541	0.0387
8192	1-way	0.0681	0.0461	0.0332
8192	2-way	0.0577	0.0402	0.0291
8192	4-way	0.0510	0.0359	0.0268
8192	8-way	0.0483	0.0339	0.0259
16384	1-way	0.0463	0.0313	0.0221
16384	2-way	0.0361	0.0248	0.0178
16384	4-way	0.0329	0.0225	0.0162
16384	8-way	0.0307	0.0207	0.0147
32768	1-way	0.0295	0.0201	0.0143
32768	2-way	0.0224	0.0153	0.0108
32768	4-way	0.0202	0.0139	0.0098
32768	8-way	0.0190	0.0132	0.0094
65536	1-way	0.0192	0.0128	0.0090
65536	2-way	0.0139	0.0092	0.0065
65536	4-way	0.0119	0.0080	0.0056
65536	8-way	0.0109	0.0074	0.0054
131072	1-way	0.0143	0.0094	0.0065
131072	2-way	0.0102	0.0064	0.0042
131072	4-way	0.0091	0.0056	0.0036
131072	8-way	0.0089	0.0054	0.0034
262144	1-way	0.0121	0.0078	0.0052
262144	2-way	0.0093	0.0057	0.0036
262144	4-way	0.0088	0.0053	0.0033
262144	8-way	0.0088	0.0052	0.0032

Table 3-5. Instruction Cache Miss Ratios.

This table shows warm-start miss ratios for instruction caches with LRU replacement and no prefetching on trace *2nd500k*.

Warm-Start Miss Ratios for Data Caches				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
256	1-way	0.3073	0.3075	0.3415
256	2-way	0.2684	0.2656	0.2815
512	1-way	0.2476	0.2444	0.2726
512	2-way	0.2094	0.2032	0.2129
512	4-way	0.1923	0.1845	0.1942
1024	1-way	0.1952	0.1877	0.1947
1024	2-way	0.1597	0.1529	0.1543
1024	4-way	0.1427	0.1358	0.1401
1024	8-way	0.1384	0.1308	0.1334
2048	1-way	0.1519	0.1423	0.1420
2048	2-way	0.1177	0.1089	0.1078
2048	4-way	0.1067	0.0991	0.0967
2048	8-way	0.1023	0.0946	0.0928
4096	1-way	0.1099	0.1010	0.1004
4096	2-way	0.0870	0.0791	0.0751
4096	4-way	0.0798	0.0730	0.0699
4096	8-way	0.0769	0.0696	0.0654
8192	1-way	0.0854	0.0762	0.0744
8192	2-way	0.0678	0.0599	0.0552
8192	4-way	0.0607	0.0528	0.0482
8192	8-way	0.0588	0.0509	0.0463
16384	1-way	0.0636	0.0564	0.0534
16384	2-way	0.0530	0.0464	0.0407
16384	4-way	0.0496	0.0418	0.0361
16384	8-way	0.0484	0.0396	0.0334
32768	1-way	0.0476	0.0425	0.0392
32768	2-way	0.0397	0.0359	0.0310
32768	4-way	0.0370	0.0342	0.0282
32768	8-way	0.0358	0.0336	0.0271
65536	1-way	0.0341	0.0288	0.0259
65536	2-way	0.0280	0.0243	0.0217
65536	4-way	0.0254	0.0224	0.0206
65536	8-way	0.0241	0.0217	0.0206
131072	1-way	0.0267	0.0211	0.0184
131072	2-way	0.0212	0.0157	0.0127
131072	4-way	0.0195	0.0142	0.0116
131072	8-way	0.0189	0.0134	0.0110
262144	1-way	0.0221	0.0161	0.0125
262144	2-way	0.0192	0.0131	0.0096
262144	4-way	0.0185	0.0123	0.0086
262144	8-way	0.0183	0.0120	0.0082

Table 3-6. Data Cache Miss Ratios.

This table shows warm-start miss ratios for data caches with LRU replacement and no prefetching on trace *2nd500k*.

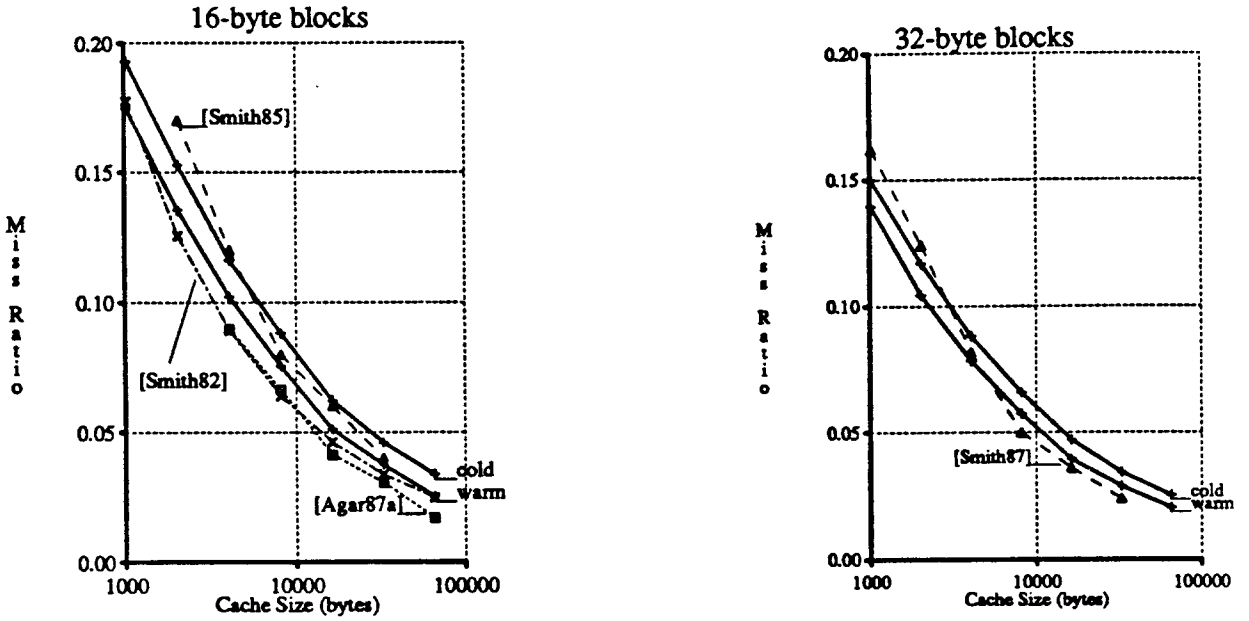


Figure 3-8. A Comparison to Other Miss Ratio Data.

This figure compares cold-start ("cold") and warm-start ("warm") miss ratios for trace *2nd500k* in eight-way set-associative mixed caches, having 16-byte (left) and 32-byte (right) blocks, to miss ratios from other sources.

Lines labeled "[Smit85]" (left) and "[Smit87]" (right) show Smith's design target miss ratios for fully-associative caches. The line labeled "[Agar87a]" (left) shows four-way set-associative miss ratio results from Figure 18 in that paper. Finally, the line labeled "[Smit82]" (also left) shows eight-way set-associative miss ratios from Figure 33 of that paper, which were gathered monitoring an Amdahl 470.

This figure shows that the miss ratios of composite trace *2nd500K* are reasonable.

Three Miss Ratio Components								
Cache Size (bytes)	Degree of Associativity	Miss Ratio	Miss Ratio Components (Relative Percent)					
			Compulsory		Capacity		Conflict	
1024	1-way	0.1913	0.0090	5%	0.1405	73%	0.0419	22%
1024	2-way	0.1609	0.0090	6%	0.1405	87%	0.0115	7%
1024	4-way	0.1523	0.0090	6%	0.1405	92%	0.0029	2%
1024	8-way	0.1488	0.0090	6%	0.1405	94%	-0.0006	-0%
2048	1-way	0.1482	0.0090	6%	0.1032	70%	0.0361	24%
2048	2-way	0.1223	0.0090	7%	0.1032	84%	0.0102	8%
2048	4-way	0.1148	0.0090	8%	0.1032	90%	0.0027	2%
2048	8-way	0.1128	0.0090	8%	0.1032	91%	0.0006	1%
4096	1-way	0.1089	0.0090	8%	0.0730	67%	0.0270	25%
4096	2-way	0.0948	0.0090	9%	0.0730	77%	0.0129	14%
4096	4-way	0.0868	0.0090	10%	0.0730	84%	0.0049	6%
4096	8-way	0.0842	0.0090	11%	0.0730	87%	0.0022	3%
8192	1-way	0.0868	0.0090	10%	0.0521	60%	0.0257	30%
8192	2-way	0.0693	0.0090	13%	0.0521	75%	0.0082	12%
8192	4-way	0.0650	0.0090	14%	0.0521	80%	0.0040	6%
8192	8-way	0.0629	0.0090	14%	0.0521	83%	0.0018	3%
16384	1-way	0.0658	0.0090	14%	0.0375	57%	0.0194	29%
16384	2-way	0.0535	0.0090	17%	0.0375	70%	0.0070	13%
16384	4-way	0.0494	0.0090	18%	0.0375	76%	0.0029	6%
16384	8-way	0.0478	0.0090	19%	0.0375	78%	0.0014	3%
32768	1-way	0.0503	0.0090	18%	0.0279	55%	0.0134	27%
32768	2-way	0.0412	0.0090	22%	0.0279	68%	0.0043	11%
32768	4-way	0.0383	0.0090	23%	0.0279	73%	0.0014	4%
32768	8-way	0.0377	0.0090	24%	0.0279	74%	0.0008	2%
65536	1-way	0.0386	0.0090	23%	0.0192	50%	0.0105	27%
65536	2-way	0.0296	0.0090	30%	0.0192	65%	0.0015	5%
65536	4-way	0.0279	0.0090	32%	0.0192	69%	-0.0002	-1%
65536	8-way	0.0275	0.0090	33%	0.0192	70%	-0.0006	-2%
131072	1-way	0.0261	0.0090	34%	0.0041	16%	0.0130	50%
131072	2-way	0.0195	0.0090	46%	0.0041	21%	0.0064	33%
131072	4-way	0.0164	0.0090	55%	0.0041	25%	0.0033	20%
131072	8-way	0.0151	0.0090	59%	0.0041	27%	0.0021	14%

Table 3-7. Three Miss Ratio Components.

This table illustrates the effect of dividing the miss ratios for trace "ue" into compulsory, capacity and conflict misses. All miss ratios are warm-start and for a mixed cache with 32-byte blocks. Under each miss ratio component, the first number is the component's absolute size, while the second is its relative contribution to the overall the miss ratio. Results for large caches are unstable, since small miss ratio variations can cause a large perturbation in a component's relative size.

3.2.2. Smith's Model of Set-Associativity

In 1978 Smith [Smit78] proposed that set-associative miss ratios can be estimated from fully-associative ones by assuming that active blocks independently map to sets and are equally likely to map to any set. This technique is not important for estimating such miss ratios, however, since all-associativity simulation allows all set-associative miss ratios to be calculated with a single simulation per cache type and block size (see Chapter 2). Rather the technique is important, because it shows that the principal cause of conflict misses is that a random mapping of active blocks to sets does not yield an exactly even mapping of active block to sets (i.e, each of s sets does not get $1/s$-th of the active blocks).

Here I show empirically that Smith model's is valid for small and large caches of any associativity including direct-mapped. Smith validated his model for caches having 64 sets and 32-byte blocks with

APL and FORTRAN traces from the IBM 360/91. Therefore, data on caches larger than 16K bytes are limited to those with associativities of eight-way and larger. His results show very good predictions for all but one case. In the final case, Smith conjectures that the predictions are pessimistic, because the simultaneous use of adjacent blocks in the simulation causes fewer collisions than were predicted by the model's random mapping of blocks to sets.

Smith's model is based on LRU stacks, which is a list of blocks from most-recently-referenced to least-recently-referenced (see Chapter 2). Recall that fully-associative caches using LRU replacement can be modeled with a single LRU stack, and that the hit ratio of an i-block fully-associative cache is the sum of the probabilities that a reference is made to distance 1 (first in the LRU stack), distance 2, ..., or distance i [Matt70]. Let the probability of finding a reference at distance i be referred to as *distance probability* d_i. Similarly, a set-associative cache with s sets can be modeled with s LRU stacks, and the hit ratio with i-way set-associativity is the sum of the probabilities of hits to any stack at distances 1 through i, each denoted with $d_i(s)$.

Smith's model estimates set-associative miss ratios from fully-associative miss ratios by estimating set-associative distance probabilities from fully-associative ones. In particular, he estimates $d_i(s)$ with Bayes Rule to be a function of d_j for $j = i, i+1, i+2, \cdots$. Bayes Rule states that the probability of some event is equal to the sum of the probabilities of that event *given* each of a number of mutually exclusive events, whose union is the sample space. Smith observes that the d_j's are probabilities of mutually exclusive events, whose union is the sample space. Therefore:

$$d_i(s) = \sum_{j=1}^{\infty} Prob(s\text{-}set\text{-}associative\ distance\ i \mid fully\text{-}associative\ distance\ j) * d_j.$$

Consequently, one can estimate set-associative distance probabilities from fully-associative ones if one can estimate $Prob(s\text{-}set\text{-}associative\ distance\ i \mid fully\text{-}associative\ distance\ j)$ for all relevant i and j. Since there are always $k-1$ blocks above a block at distance k in an LRU stack, another way of stating the above probability is as the probability that $i-1$ blocks are above of a reference in s LRU stacks given $j-1$ blocks are above it is a single LRU stack. Thus, finding $Prob(s\text{-}set\text{-}associative\ distance\ i \mid fully\text{-}associative\ distance\ j)$ reduces to calculating how often exactly $i-1$ of the $j-1$ more recently-referenced blocks map to the same set as the current reference. Clearly, this probability is zero if $j<i$, since there are not $i-1$ more recently-referenced blocks that could map to the set of a reference. If $j \geq i$ on the other hand, Smith makes the simplest assumption, namely, that active blocks independently map to one of s sets with probability $1/s$. The assumption is not strictly true in practice because of spatial locality, which makes it more likely that the recently-referenced blocks map to adjacent sets. Nevertheless, the assumption is useful for predicting set-associative behavior. The probability that a more recently-referenced block maps to the same set as the referenced block is $1/s$, because the independence assumption applies to both blocks. Since all the more recently-referenced blocks are independent of each other, the probability that exactly $i-1$ of the $j-1$ more recently-referenced blocks are in the set of the reference can be estimated with a binomial distribution to be:

$$Prob(s\text{-}set\text{-}associative\ distance\ i \mid fully\text{-}associative\ distance\ j) \approx$$
$$\binom{j-1}{i-1} \left(\frac{1}{s}\right)^{i-1} \left(\frac{s-1}{s}\right)^{j-i};\ j \geq i$$

Therefore, by Bayes rule:

$$\hat{a}_i(s) = \sum_{j=i}^{\infty} \binom{j-1}{i-1} \left(\frac{1}{s}\right)^{i-1} \left(\frac{s-1}{s}\right)^{j-i} * d_j$$

where $\hat{a}_i(s)$ denotes Smith's estimate of $d_i(s)$.

Some intuition for this approximation can be gained by examining its prediction for the hit ratio of an s-block direct-mapped cache, $d_1(s)$:

$$\hat{a}_1(s) = d_1 + \left(\frac{s-1}{s}\right) d_2 + \left(\frac{s-1}{s}\right)^2 d_3 + \cdots + \left(\frac{s-1}{s}\right)^{j-1} d_j + \cdots$$

This equation predicts a hit if the block referenced is the most-recently-referenced block, or if it is the second most-recently-referenced block and the most-recently-referenced block maps to another set, or if it is the third most-recently-referenced block and the first and second most-recently-referenced blocks map to other sets, etc.

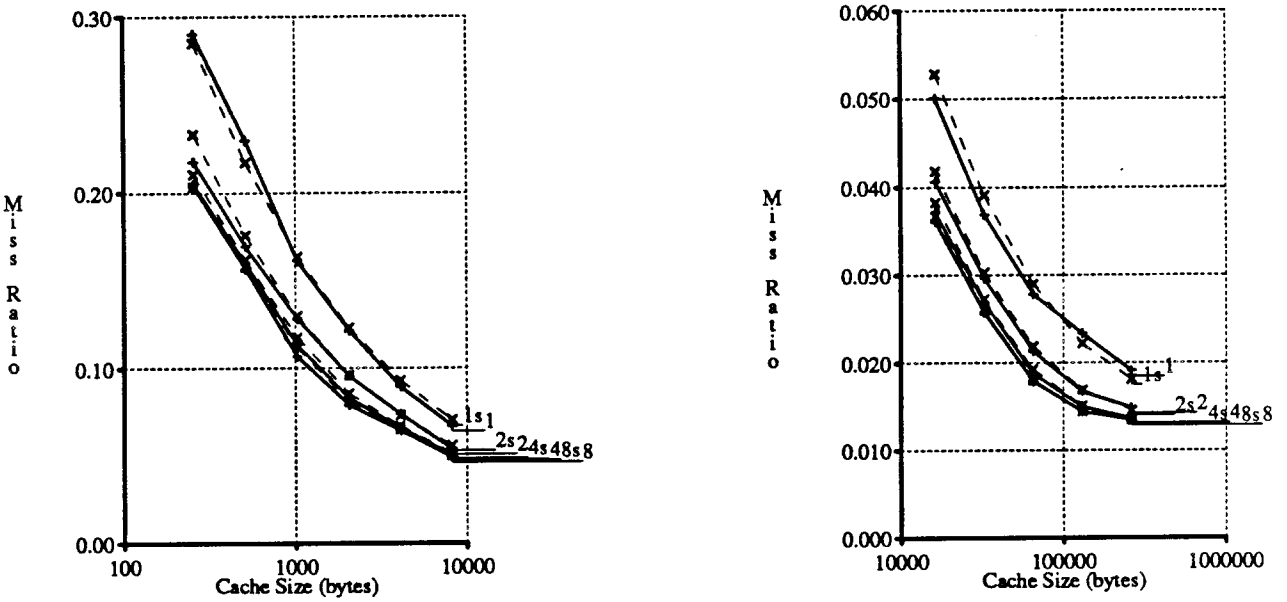


Figure 3-9. Smith's Predictions for "mul2".

This figure shows Smith's predictions (dashed lines, labeled with 's') and actual set-associative miss ratios with trace *mul2* for mixed caches with 32-byte blocks. All predictions are accurate, particularly those for higher associativities.

Figures 3-9 and 3-10 show set-associative miss ratios (solid lines) and Smith's predictions (dashed lines). These figures and predictions for other traces (not shown) yield three conclusions:

(1) Smith's predictions are accurate. Let the relative error between Smith's prediction and an actual miss ratio be the predicted miss ratio less the actual miss ratio, all divided by the actual miss ratio. In most cases the relative error is less than five percent, and only rarely is it greater than ten percent.

(2) Smith's predictions tend to be more pessimistic than actual miss ratios (see Figure 3-10). I agree with Smith that this is because spatial locality makes blocks selected with bit selection less likely to collide than blocks selected with random mapping.

(3) The relative error gets smaller as associativity is increased, which I expected, since many-way set-associative caches have miss ratios nearly identical to fully-associative caches.

3.2.3. Ratios of Set-Associative Miss Ratios

In the last sections we saw that many factors affect the miss ratio of set-associative caches. Next I isolate the effects of associativity by examining the *relative* effect of changing associativity in simulations based on individual traces and averages of traces.

A new metric I use here is *miss ratio spread*, which between an n-way set-associative cache, $C(A=n)$, and a similar $2n$-way set-associative cache, $C(A=2n)$, is the ratio of the miss ratios, less one, or:

$$\frac{m(C(A=n))}{m(C(A=2n))} - 1 = \frac{m(C(A=n)) - m(C(A=2n))}{m(C(A=2n))}$$

where $m(C(A=i))$ is the miss ratio of an i-way set-associative cache. The use of miss ratio spread

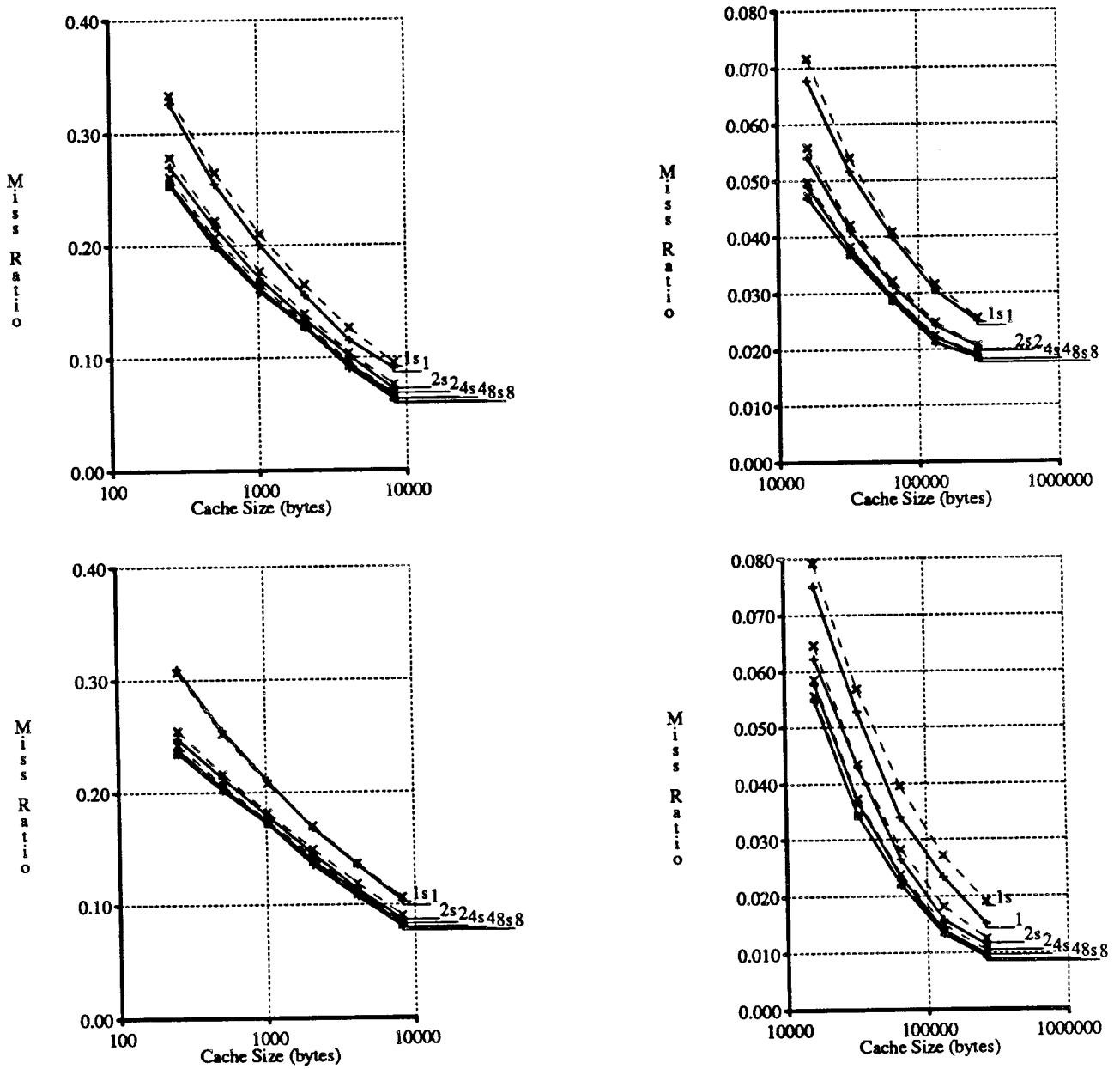


Figure 3-10. Smith's Miss Ratio Predictions.

For "ue" (top) and "mvs" (bottom).

facilitates examining the effects of associativity by reducing the effects of other parameters, like cache size and block size.

Figures 3-11 and 3-12 show miss ratio spreads for various traces (with cold-start, mixed cache, 32-byte blocks and LRU replacement). I smooth the data with a weighted average of adjacent spreads (recommended in [Cham83]). Let $mrs(c)$ be the miss ratio spread for a cache of c blocks. The smoothed spread, $mrs'(c)$, is $0.15*mrs(c/4) + 0.20*mrs(c/2) + 0.30*mrs(c) + 0.20*mrs(2c) + 0.15*mrs(4c)$. I selected weights to reduce variation between adjacent spreads, without suppressing larger trends. End points are estimated using remaining points with weights increased proportionally to sum to 1.0. For example, for a 1K-byte cache, $mrs'(1K)$, is $[0.30*mrs(1K) + 0.20*mrs(2K) + 0.15*mrs(4K)]/[1.0-0.15-0.30]$.

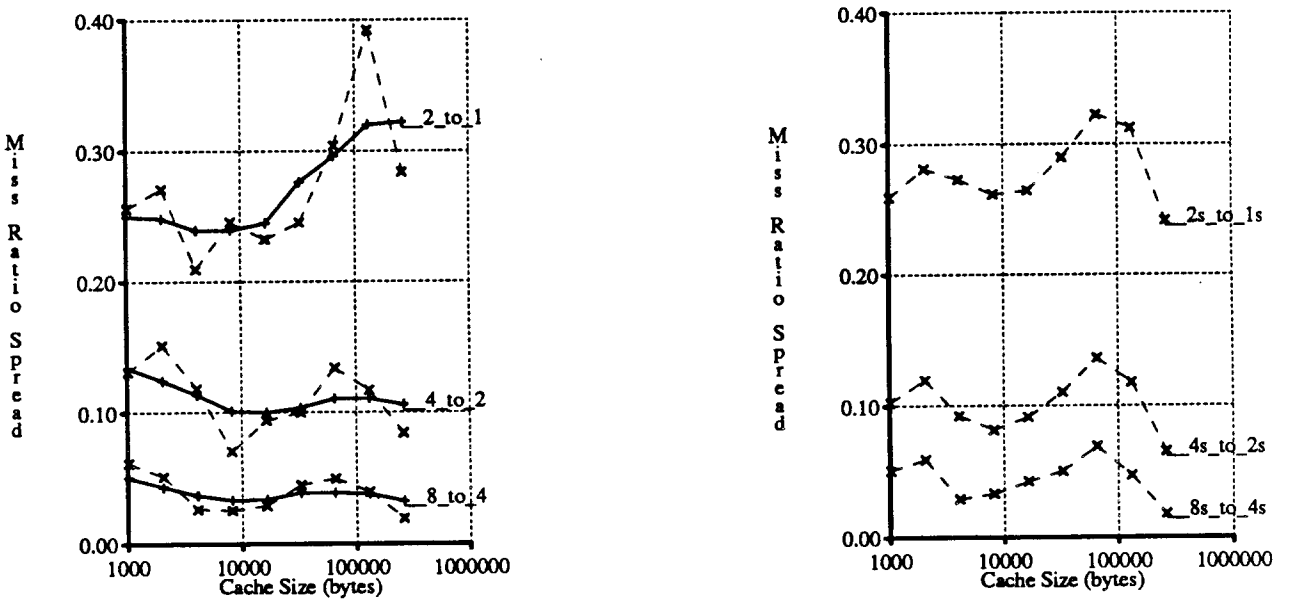


Figure 3-11. Cold-Start Miss Ratio Spreads for "mul2".

This figure shows cold-start miss ratios spreads (the ratio of miss ratios of caches with different associativity, less one) for mixed caches using 32-byte blocks and LRU replacement with trace *mul2*. The left plot shows smoothed (solid lines) and actual miss ratios spreads (dashed lines). Each smoothed value is the weighted average of the actual value and the actual values to two points below and above (see text for a more exact description of data smoothing). The data imply that the miss ratio spreads are smaller between caches of greater associativity and that there is no strong relationship between miss ratio spread and cache size.

The right plot displays miss ratios spreads of set-associative miss ratios calculated from fully-associative miss ratios using Smith's method. It shows that Smith's predictions yield miss ratios spreads that are quite similar to actual miss ratio spreads, which illustrates that even small relative miss ratio changes are explained by Smith's model.

Quantitative results vary, but two qualitative trends are present. First, as evident in the original miss ratios, the miss ratio spread between direct-mapped and two-way set-associative caches is greater than that between two-way and four-way caches, which is greater than that between four-way and eight-way caches. Second and not so obvious in the original miss ratios, there is no general correlation between miss ratio spread and cache size.

To study the effects of cache type and block size on miss ratio spread, I now restrict simulation analysis to the composite trace *2nd500k*. Figures 3-13 and 3-14 show cold- and warm-start miss ratio spreads for mixed, instruction and data caches with 32-byte blocks. Since warm-start miss ratio spreads are qualitatively similar to cold-start spreads, and warm-start spreads are arguably more realistic, I use only warm-start numbers in the rest of this section. I calculate the miss ratio spread with the *ratios of averages*, defined in Section 3.1.2.

Miss ratios spreads show surprisingly little systematic variation with changing cache size. For this reason I can reduce data further by averaging miss ratio spreads for caches of different sizes. Average miss ratios spreads are presented in Table 3-8. Table 3-9 shows the same data expressed in miss ratio change relative to direct-mapped miss ratios. The miss ratio spreads between direct-mapped and two-set-associative instruction caches, however, do show systematic variation with cache size (see Figure 3-14). For this reason averaging across varying instruction cache sizes can be misleading.

The data in Table 3-8 exhibit several trends.

(1) As is known, miss ratio spreads are larger between caches of more restricted associativity. Mixed caches with 32-byte blocks, for example, have miss ratio spreads of 4, 10 and 25 percent as associativity is successively halved from eight-way.

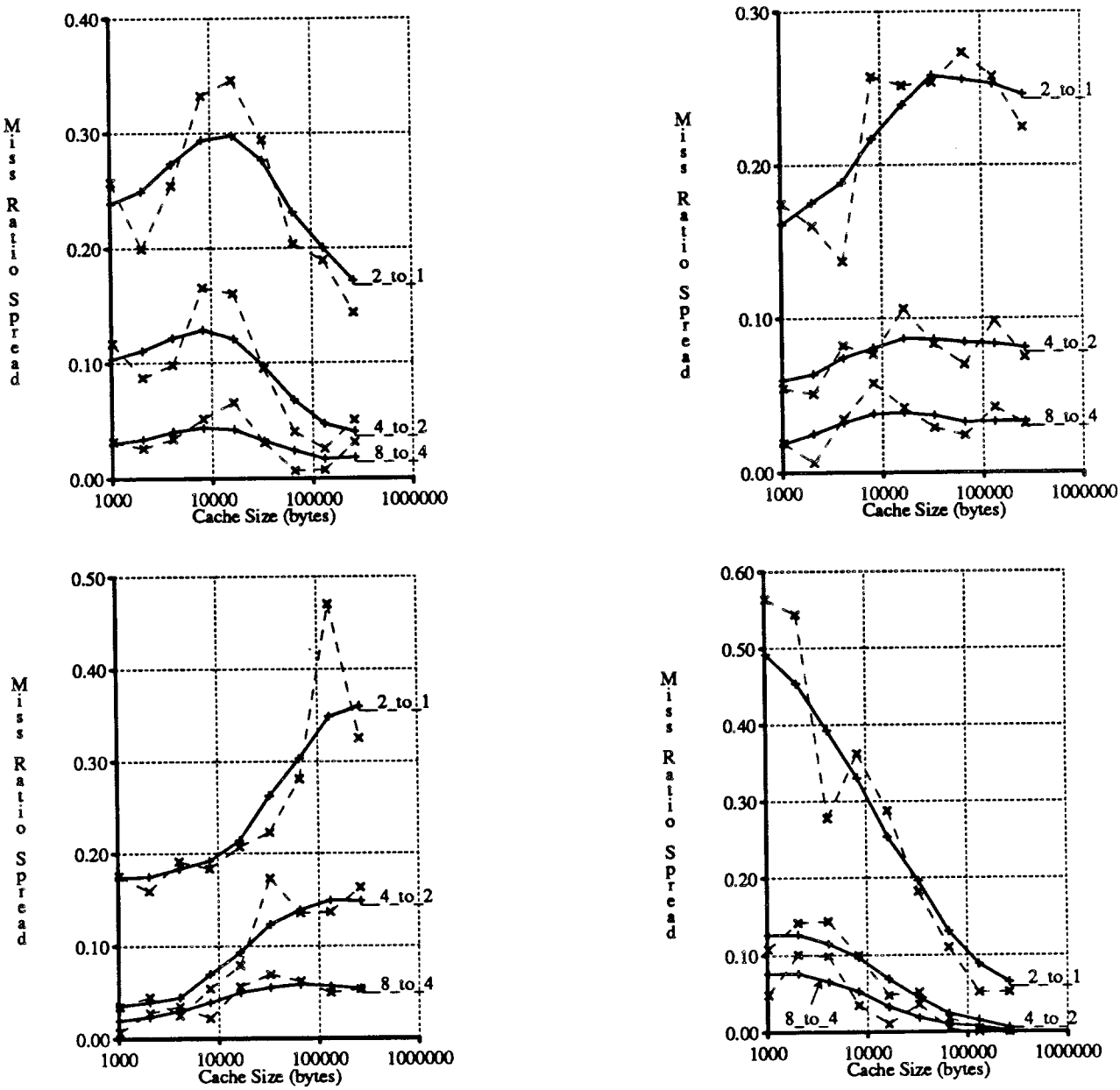


Figure 3-12. Cold-start Miss Ratios Spreads.

For "mul8" (top-left), "ue" (top-right), "mvs" (bottom-left) and "synapse_devel" (bottom-right).

(2) Miss ratio spreads do not vary much with changing cache type. Halving associativity in two-way set-associative caches with 32-byte blocks, for instance, increases miss ratio 25 percent, regardless of cache type. Results with other traces, however, suggest that some instruction cache miss ratio spreads are lower than those for mixed and data caches (see Table 3-10 and [Cho86]).

(3) Miss ratio spreads are positively correlated with block size. Halving associativity in two-way mixed set-associative caches, for example, increases the miss ratio 22, 25 and 32 percent with 16, 32 and 64-byte blocks.

The above miss ratio spreads were calculated with *ratio of averages* rather than with the *average of ratios*. These methods are distinguished in Section 3.1.2. I now explore whether there is any significant difference between miss ratio spreads computed with the two methods using warm-start miss ratios from 23 of 33 trace samples gathered by Agarwal, Sites and Horowitz and distributed by DEC

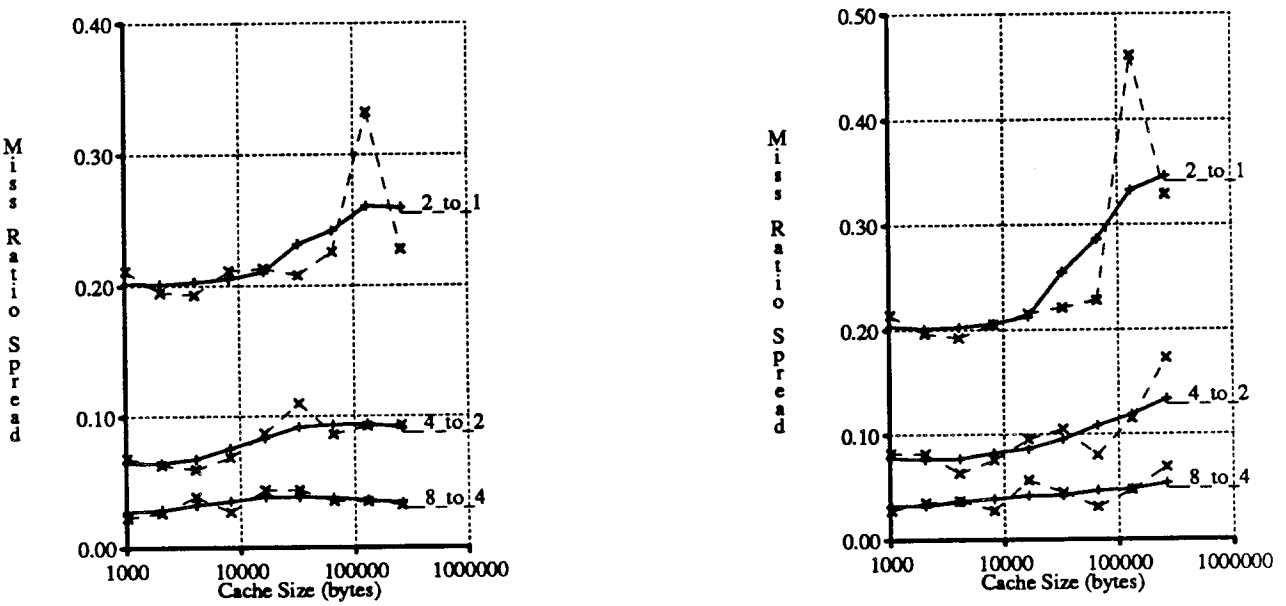


Figure 3-13. Mixed Miss Ratio Spreads for "2nd500k".

This figure shows cold- and warm-start miss ratio spreads for mixed caches using 32-byte blocks and LRU replacement with trace *2nd500k*, whose miss ratios are the arithmetic average of the miss ratio of several traces.

For the most part, miss ratio spreads vary little with changing cache size. The only major exception to this rule is the miss ratio spread between direct-mapped and two-way set-associative 128K-byte caches. I expect the cause of this aberration lies in the particular traces and trace lengths used, not in some property of 128K-byte caches. No such aberration is present, for example, in the miss ratio spreads of Figure 3-15. Warm-start miss ratios spreads are on balance greater, but the difference is often small and sometimes negative.

[Agar86]. I use a different set of traces, which I will call *atum*, than before to see if additional results will corroborate the miss ratios spreads found with *2nd500k*.

Figure 3-15 and Table 3-10 show miss ratio spreads for *atum*. Since results for the two methods are nearly identical, either method may be used. Results here are also comparable to miss ratio spreads found with *2nd500k*. The spreads between two-way set-associative and direct-mapped mixed and data caches, however, are a little higher here than they were with *2nd500k* (27 to 31 percent vs. 25 percent); while spreads between two-way set-associative and direct-mapped instructions caches are 21 to 25 percent here and 25 percent for *2nd500k*.

I conclude this section by extending Smith's *design target miss ratios* [Smit85, Smit87] to caches of more restricted associativity. I use Smith's fully-associativity miss ratios to approximate eight-way set-associative miss ratios, and compute other set-associative miss ratios using smoothed miss ratio spreads calculated with the average of ratios for *atum*. Tables 3-12, 3-13 and 3-14 show my extended design target miss ratios.

In this section I have examined the relationship between the miss ratios of caches with different associativities, and find that the relative difference, the miss ratio spread, does not change dramatically as caches get larger. Consequently the absolute miss ratio difference decreases as cache gets larger, since each miss ratio gets smaller. The next chapter shows that when the absolute miss ratio difference becomes sufficiently small, an interesting change occurs: the effective access time of a direct-mapped cache can be smaller than that of a set-associative cache of the same size, even though the direct-mapped cache has the larger miss ratio. This change occurs when implementation differences, that have previously been ignored, become more important than absolute miss ratio differences.

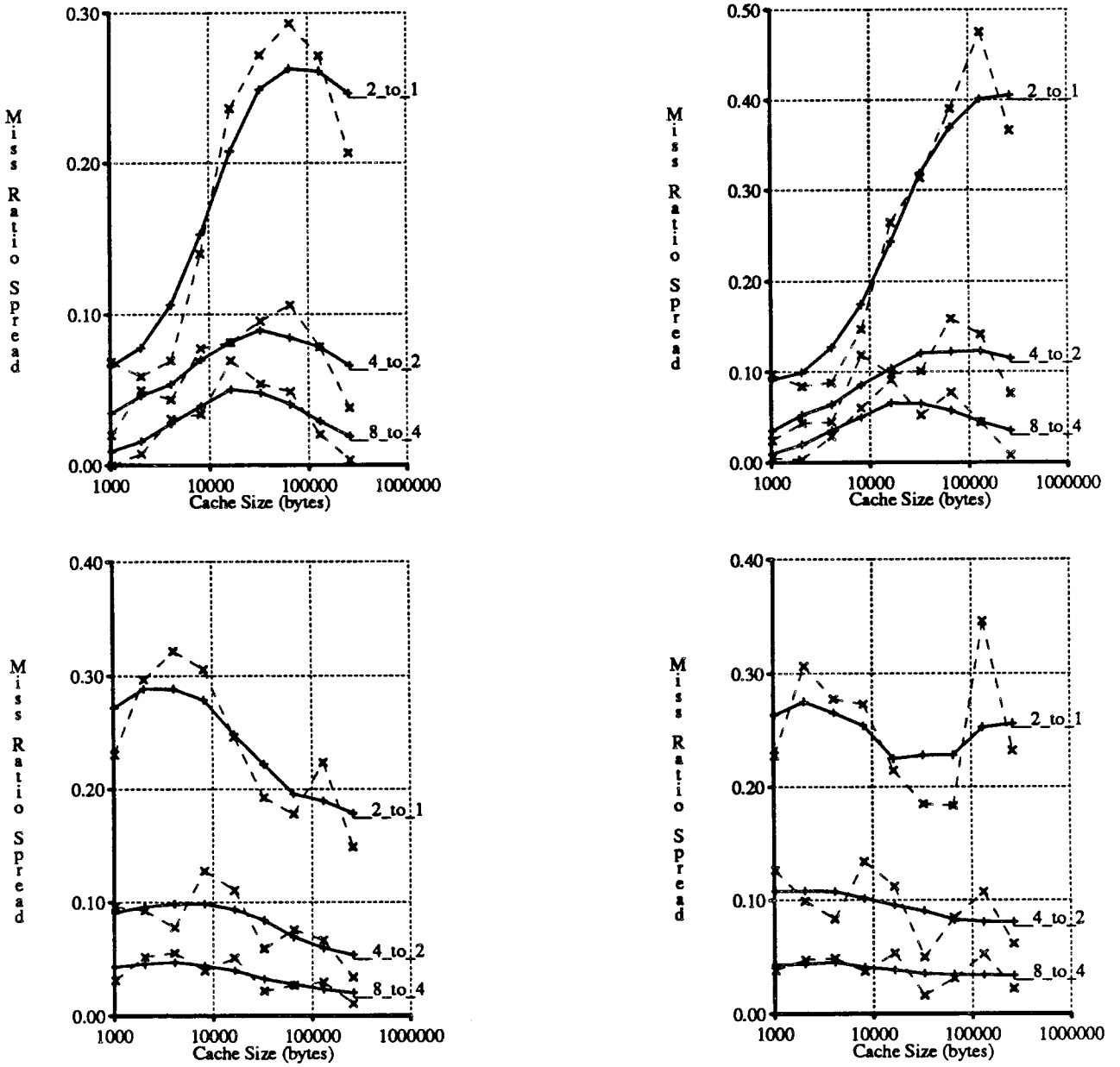


Figure 3-14. Miss Ratio Spreads for "2nd500k".

Cold-start (left), warm-start (right), instruction caches (top) and data caches (bottom).

Miss Ratio Spreads w/ 16-byte Blocks			
Cache Type	Associativity Change		
	8-to-4	4-to-2	2-to-1
Mixed	4%	9%	22%
Instruction	4%	9%	23%*
Data	3%	9%	23%

Miss Ratio Spreads w/ 32-byte Blocks			
Cache Type	Associativity Change		
	8-to-4	4-to-2	2-to-1
Mixed	4%	10%	25%
Instruction	4%	9%	25%*
Data	4%	10%	25%

Miss Ratio Spreads w/ 64-byte Blocks			
Cache Type	Associativity Change		
	8-to-4	4-to-2	2-to-1
Mixed	4%	11%	32%
Instruction	4%	9%	27%*
Data	5%	10%	31%

Miss Ratio Spread SUMMARY			
Block Size (bytes)	Associativity Change		
	8-to-4	4-to-2	2-to-1
16	4%	9%	23%
32	4%	10%	25%
64	4%	10%	30%

Table 3-8. Average Miss Ratio Spreads for "2nd500k".

The top three tables summarize average warm-start miss ratio spreads for various types of caches with various block sizes. The average is taken across miss ratio spreads for 1K-byte to 256K-byte caches. The average is meaningful when miss ratios spreads do not change systematically with cache size. Figure 3-14 shows, however, that the warm-start miss ratio spread between a two-way set-associative and a direct-mapped instruction caches vary from 10 to 40 percent as cache size increases from 1K bytes to 256K bytes. For this reason, these averages, denoted with asterisks in the above table, are suspect.

The final table collapses results from different cache types (mixed, instruction, and data) to give overall miss ratio spreads that are only functions of associativity and block size. This compression is reasonable since the data show little variation with changing cache type. Other data, however, in Table 3-10 and in [Cho86] suggest that instruction spreads can be smaller than those for mixed and data caches.

Vs. DM Miss Ratios w/ 16-byte Blocks			
Cache Type	Associativity Change		
	1-to-8	1-to-4	1-to-2
Mixed	-27%	-24%	-18%
Instruction	-27%	-24%	-18%
Data	-28%	-25%	-19%

Vs. DM Miss Ratios w/ 32-byte Blocks			
Cache Type	Associativity Change		
	1-to-8	1-to-4	1-to-2
Mixed	-30%	-27%	-20%
Instruction	-28%	-25%	-19%
Data	-29%	-27%	-20%

Vs. DM Miss Ratios w/ 64-byte Blocks			
Cache Type	Associativity Change		
	1-to-8	1-to-4	1-to-2
Mixed	-34%	-31%	-24%
Instruction	-29%	-26%	-20%
Data	-34%	-30%	-23%

Vs. DM Miss Ratios SUMMARY			
Block Size (bytes)	Associativity Change		
	1-to-8	1-to-4	1-to-2
16	-27%	-24%	-18%
32	-29%	-26%	-20%
64	-32%	-29%	-22%

Table 3-9. Relative to DM Miss Ratios for "2nd500k".

The top three tables present the data of Table 3-8 is a slightly different manner. Instead of showing the relative miss ratio change from doubling associativity, the miss ratio spread, this table shows the relative miss ratio change that results from increasing associativity from one. All changes are negative since set-associative caches have smaller miss ratios than direct-mapped (DM) caches.

The final table collapses results from different cache types (mixed, instruction, and data) to give overall numbers that are only functions of associativity and block size.

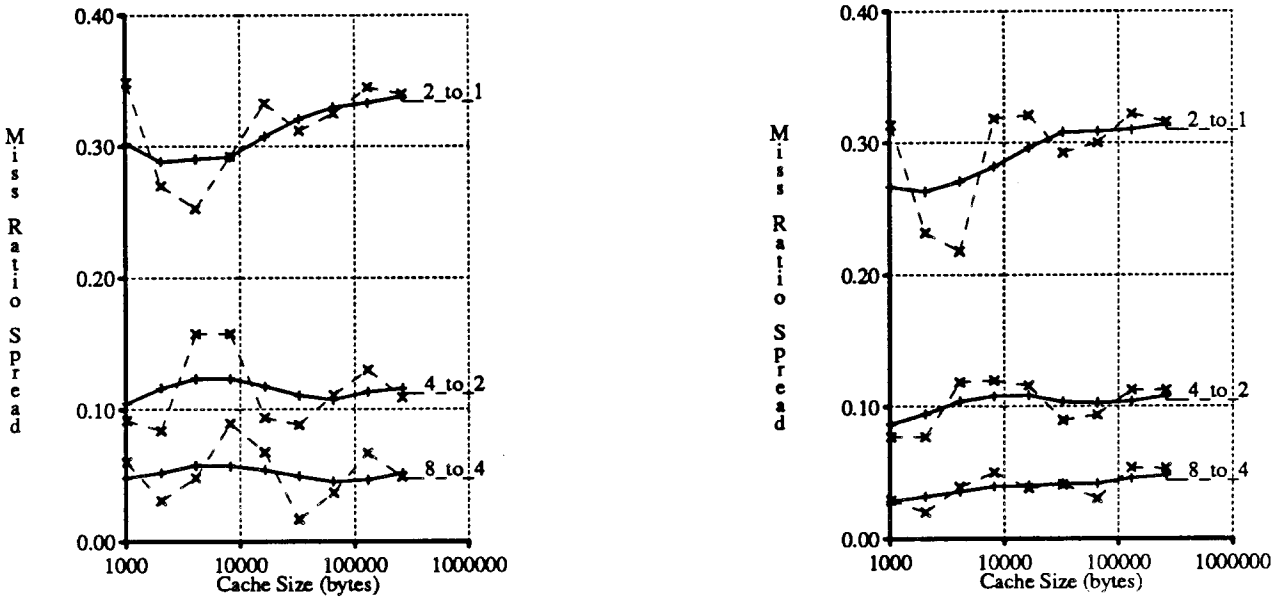


Figure 3-15. Miss Ratio Spreads for "atum".

This figure shows warm-start miss ratio spreads for mixed caches using 32-byte blocks and LRU replacement with *atum*. Spreads are calculated as the ratios of averages (left) and average of ratios (right). Solid lines show data smoothed as before.

Miss Ratio Spreads *(average of ratios)*			
Cache Type	Associativity Change		
	8-to-4	4-to-2	2-to-1
Mixed	5%	11%	31%
Instruction	5%	13%	25%
Data	5%	11%	29%

Miss Ratio Spreads *(ratio of the averages)*			
Cache Type	Associativity Change		
	8-to-4	4-to-2	2-to-1
Mixed	4%	10%	29%
Instruction	3%	9%	21%
Data	4%	9%	27%

Table 3-10. Average Miss Ratio Spreads for "atum".

These tables show average warm-start miss ratio spreads with 32-byte blocks and *atum*, calculated as the ratios of averages (top) and average of ratios (bottom).

Vs. DM Miss Ratios *(average of ratios)*			
Cache Type	Associativity Change		
	1-to-8	1-to-4	1-to-2
Mixed	-35%	-32%	-24%
Instruction	-32%	-28%	-20%
Data	-33%	-30%	-22%

Vs. DM Miss Ratios *(ratio of the average)*			
Cache Type	Associativity Change		
	1-to-8	1-to-4	1-to-2
Mixed	-32%	-30%	-23%
Instruction	-26%	-24%	-17%
Data	-30%	-28%	-21%

Table 3-11. Relative to DM Miss Ratios for "atum".

These tables re-present the data of Table 3-10 by showing the relative miss ratio change resulting from increasing associativity from one (DM), instead of miss ratio spreads.

Design Target Miss Ratios for Mixed Caches w/ 16-byte Blocks				
Cache Size	Associativity			
	8-way	4-way	2-way	DM
1024	0.210	0.219	0.239	0.288
2048	0.170	0.179	0.197	0.240
4096	0.120	0.126	0.140	0.172
8192	0.080	0.084	0.093	0.116
16384	0.060	0.063	0.069	0.088
32768	0.040	0.042	0.046	0.059

Design Target Miss Ratios for Instruction Caches w/ 16-byte Blocks				
Cache Size	Associativity			
	8-way	4-way	2-way	DM
1024	0.200	0.211	0.234	0.271
2048	0.150	0.159	0.179	0.210
4096	0.100	0.106	0.120	0.143
8192	0.060	0.064	0.072	0.089
16384	0.050	0.053	0.060	0.076
32768	0.030	0.032	0.036	0.046

Design Target Miss Ratios for Data Caches w/ 16-byte Blocks				
Cache Size	Associativity			
	8-way	4-way	2-way	DM
1024	0.160	0.170	0.192	0.244
2048	0.120	0.127	0.143	0.183
4096	0.100	0.106	0.117	0.148
8192	0.080	0.084	0.092	0.116
16384	0.060	0.062	0.068	0.084
32768	0.040	0.041	0.045	0.055

Table 3-12. Design Target Miss Ratios for 16-byte Blocks.

In this table I extend Smith's design target miss ratios [Smit85, Smit87] to caches of varying associativity by multiplying Smith's numbers by miss ratio spreads calculated from the average of ratios with *atum* (such as spreads in the left-hand plot of Figure 3-15).

These miss ratios may serve as "rules of thumb" for cache designers working with "a 32-bit architecture running fairly large programs and mature (i.e., large) operating system.

Design Target Miss Ratios for Mixed Caches w/ 32-byte Blocks				
Cache Size	Associativity			
	8-way	4-way	2-way	DM
1024	0.162	0.170	0.188	0.244
2048	0.124	0.130	0.146	0.188
4096	0.082	0.087	0.097	0.126
8192	0.050	0.053	0.059	0.077
16384	0.036	0.038	0.042	0.055
32768	0.024	0.025	0.028	0.037

Design Target Miss Ratios for Instruction Caches w/ 32-byte Blocks				
Cache Size	Associativity			
	8-way	4-way	2-way	DM
1024	0.134	0.140	0.155	0.179
2048	0.098	0.103	0.117	0.138
4096	0.063	0.067	0.076	0.091
8192	0.037	0.039	0.045	0.056
16384	0.029	0.031	0.035	0.045
32768	0.017	0.018	0.021	0.027

Design Target Miss Ratios for Data Caches w/ 32-byte Blocks				
Cache Size	Associativity			
	8-way	4-way	2-way	DM
1024	0.138	0.146	0.166	0.216
2048	0.094	0.101	0.114	0.149
4096	0.070	0.075	0.084	0.109
8192	0.053	0.056	0.062	0.081
16384	0.039	0.041	0.045	0.058
32768	0.025	0.026	0.028	0.037

Table 3-13. Design Target Miss Ratios for 32-byte Blocks.

Design Target Miss Ratios for Mixed Caches w/ 64-byte Blocks				
Cache Size	Associativity			
	8-way	4-way	2-way	DM
1024	0.137	0.144	0.162	0.229
2048	0.098	0.104	0.118	0.163
4096	0.059	0.063	0.072	0.099
8192	0.033	0.035	0.040	0.055
16384	0.023	0.025	0.028	0.038
32768	0.014	0.015	0.017	0.023

Design Target Miss Ratios for Instruction Caches w/ 64-byte Blocks				
Cache Size	Associativity			
	8-way	4-way	2-way	DM
1024	0.098	0.104	0.115	0.133
2048	0.068	0.072	0.082	0.097
4096	0.043	0.046	0.053	0.063
8192	0.023	0.025	0.028	0.035
16384	0.018	0.019	0.022	0.029
32768	0.010	0.011	0.012	0.016

Design Target Miss Ratios for Data Caches w/ 64-byte Blocks				
Cache Size	Associativity			
	8-way	4-way	2-way	DM
1024	0.140	0.150	0.170	0.227
2048	0.083	0.089	0.102	0.138
4096	0.054	0.058	0.067	0.090
8192	0.039	0.042	0.047	0.064
16384	0.026	0.028	0.031	0.042
32768	0.017	0.018	0.020	0.027

Table 3-14. Design Target Miss Ratios for 64-byte Blocks.

3.3. Analysis with Effective Access Time

In this section I analyze caches with the metric effective access time to show that the effective access times of large direct-mapped caches are often smaller than those of slower set-associative caches of the same size. Consider a direct-mapped cache, C_1, and a set-associative cache, C_2, where the direct-mapped cache has a larger miss ratio ($m(C_1) > m(C_2)$), but a smaller access time ($t_{cache}(C_1) < t_{cache}(C_2)$). I model effective access time as:

$$t_{eff}(C) = t_{cache}(C) + m(C)*t_{memory}(C)$$

where $m(C)$, $t_{cache}(C)$ and $t_{memory}(C)$ are the miss ratio, cache access time and cache miss penalty in a system with cache C. If I also assume that t_{memory} is the same for both caches, then the relationship between their effective access times is:

$$t_{cache}(C_1) + m(C_1)*t_{memory} \stackrel{?}{=} t_{cache}(C_2) + m(C_2)*t_{memory}$$

where $\stackrel{?}{=}$ means either or neither side can be larger. Re-arranging terms yields:

$$(m(C_1)-m(C_2))*t_{memory} \stackrel{?}{=} t_{cache}(C_2) - t_{cache}(C_1),$$

or

$$\Delta m*t_{memory} \stackrel{?}{=} \Delta t_{cache}.$$

Therefore, a direct-mapped cache's effective access time is smaller than that of a set-associative cache if the difference in their miss ratios times the miss penalty is less the the difference in their access times. Figure 3-16 illustrates this relationship.

To see if any practical direct-mapped caches have effective access times smaller than those of set-associative caches, I must find practical values for miss ratio differences and access time differences. I next use the miss ratios gathered in the last section to find reasonable miss ratio differences; then define a cache architecture and implement it in three technologies to find some practical access time differences; and finally I combine these assumptions to show that practical direct-mapped caches can have smaller effective access times than set-associative caches of the same size.

3.3.1. Incorporating Previous Miss Ratio Analysis

Figure 3-16 shows that direct-mapped caches can have lower effective access times than set-associative caches for appropriate miss ratio differences and access time differences. In Section 3.2 I studied cache miss ratios with trace-driven simulation. I can use these miss ratios now to find example values for miss ratios differences.

Figures 3-17 and 3-18 illustrate warm-start miss ratio differences with trace *2nd500k*. The data show that, except for small instruction caches, the absolute miss ratio difference between caches of various associativities gets smaller as cache size increases. Consequently, I expect that direct-mapped caches may to do better as cache sizes increase.

Since the miss ratio difference is a function of cache size, I can incorporate it into a graph like Figure 3-16 by replacing the axis labeled "Miss Ratio Difference" with one labeled "Cache Size," and by shifting parameters around to facilitate presentation. This data is displayed in Figures 3-19, 3-20, 3-21 and 3-22.

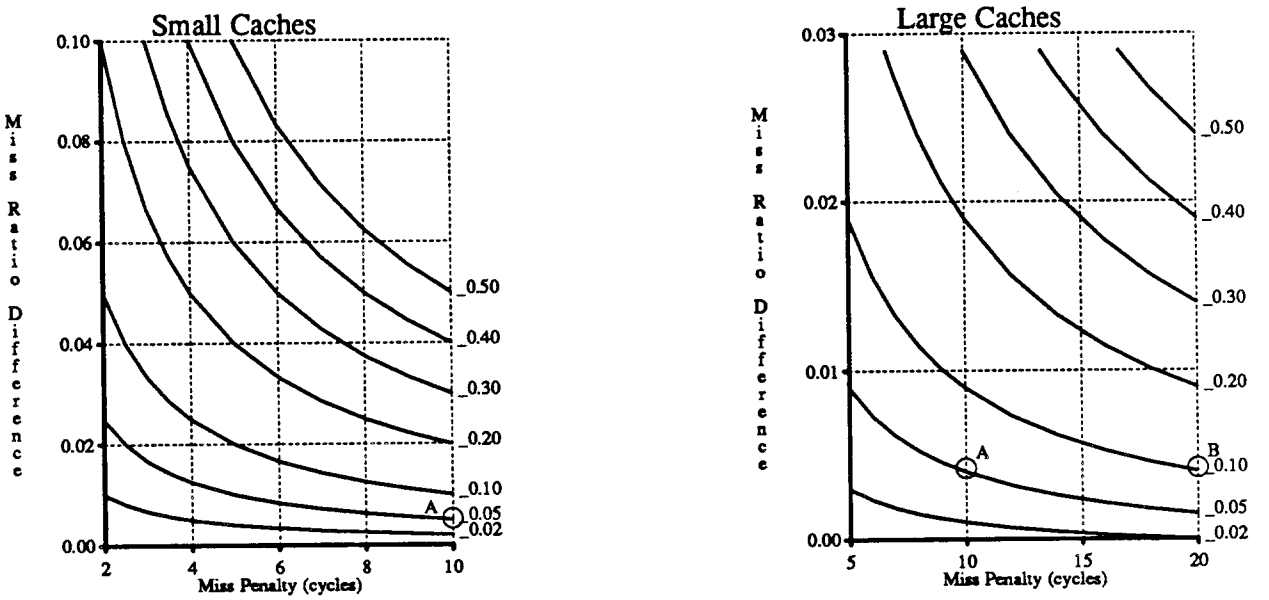


Figure 3-16. Equal Effective Access Times.

This figure illustrates the relationship between the effective access times of a direct-mapped cache, C_1, and a set-associative cache, C_2, where the direct-mapped cache has a larger miss ratio but a smaller access time than the set-associative cache, and both caches suffer the same penalty on a miss. The x-axes show various miss penalties, t_{memory}; the y-axes show various miss ratio differences, $\Delta m = m(C_1) - m(C_2)$; and the lines are labeled with values of the access time difference, $\Delta t_{cache} = t_{cache}(C_2) - t_{cache}(C_1)$ with $t_{cache}(C_1)$ defined to be 1.0 (cycle), for which the effective access times are equal, i.e., $t_{eff}(C_1) = t_{eff}(C_2)$. The relationship displayed is:

$$\Delta m * t_{memory} = \Delta t_{cache}.$$

The left plot illustrates small caches with large miss ratio differences, while the right plot shows larger caches with lower miss ratios differences. For a particular cache, Δm and t_{memory} define a point in one of the plots that can be translated into a Δt_{cache} by interpolating. A direct-mapped cache is preferred for implementations with an access time difference greater than that Δt_{cache}.

The points labeled "A" and "B" give the location of a direct-mapped 128K-byte mixed cache with 32-byte blocks using warm-start miss ratios from trace *2nd500K*, assuming a fast ten-cycle miss penalty ("A") and a slow twenty-cycle miss penalty ("B"). For a cache at design point "A," these plots imply that a direct-mapped cache has a better effective access time than a set-associative cache of the same size if their access time difference exceeds 5 percent.

Results show that direct-mapped cache performance improves with respect to set-associative cache performance as cache size increases, if the access time difference does not change. Consider a mixed cache with 32-byte blocks and a 20-cycle miss penalty. An 8K-byte direct-mapped cache has a smaller effective access time than a two-way set-associative 8K-byte cache only if the set-associative cache is at least 26 percent slower on cache hits; at 64K-bytes, the crossover occurs at 10 percent slower.

In the rest of this chapter I examine cache implementations to see if practical set-associative caches can have access times sufficiently slower than direct-mapped access times to make their effective access times larger than those of the direct-mapped caches.

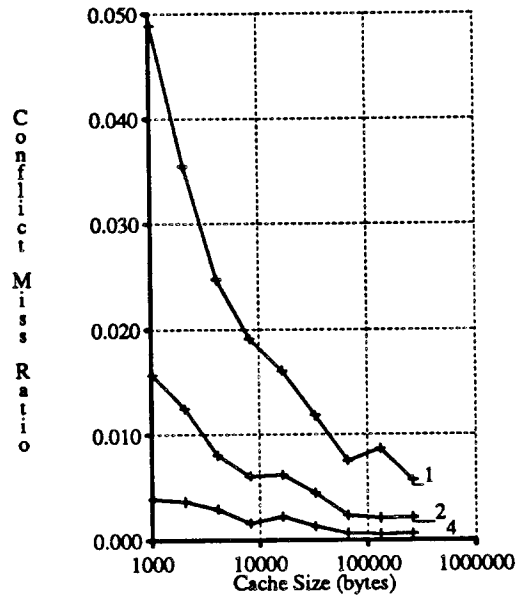


Figure 3-17. Mixed Cache Miss Ratio Differences.

The figures shows the difference between miss ratio for caches for various associativities ("1," "2" and "4") and eight-way set-associative miss ratios. The y-axis is labeled "Conflict Miss Ratio" to emphasize that direct-mapped, two-way set-associative, and four-way set-associative miss ratios are larger than eight-way set-associative miss ratios, because two many active blocks maps to some sets (see Section 3.2.1). All caches are mixed and have 32-byte blocks; miss ratios are warm-start miss ratios from trace-driven simulation with trace *2nd500k*. Miss ratio difference between caches of varying associativity are represented as the distance between lines. The miss ratio difference between direct-mapped and two-way set-associative caches, for example, is simply the difference between corresponding points on the lines labeled "1" and "2." On average, this distance decreases are cache sizes increases.

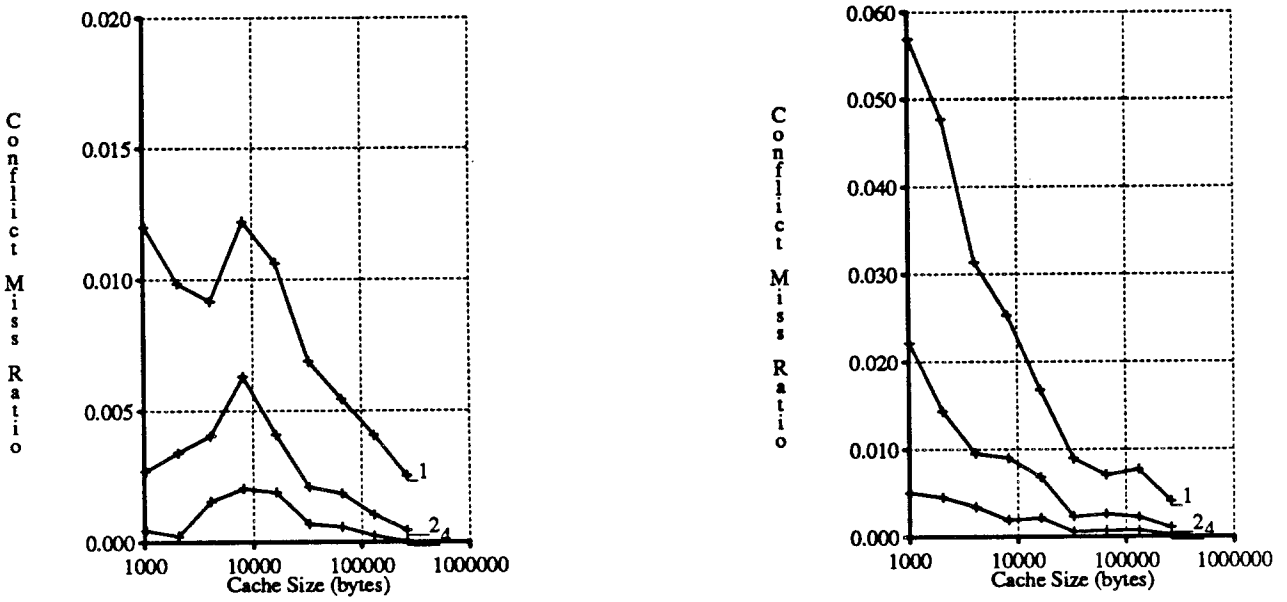


Figure 3-18. More Miss Ratio Differences.

This figure presents miss ratio differences for instruction (left) and data (right) caches. See the caption of Figure 3-17 for how this data is derived. Note that the y-axis scale for instruction caches is significantly different than that for mixed or data caches. I expected instruction caches to have smaller conflict miss ratios, since instruction references exhibit greater spatial locality. It is interesting, however, that the miss ratio differences for these instruction caches do not monotonically decrease with cache size. It remains be to seen whether this behavior in an anomaly.

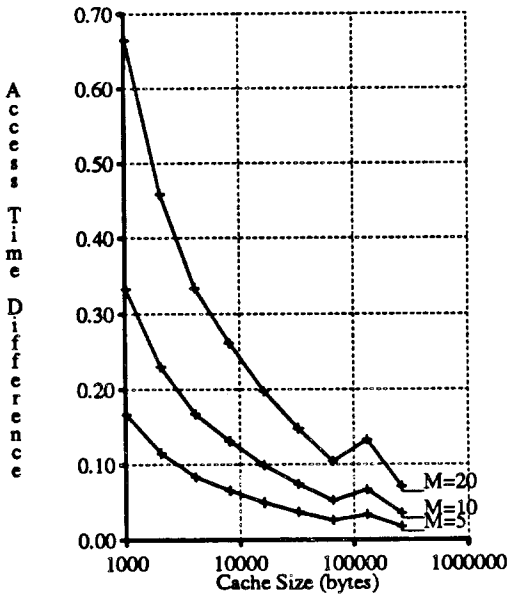


Figure 3-19. DM vs. 2-Way Mixed Cache Crossovers.

This figure combines the equation presented in Figure 3-16 with warm-start miss ratio differences shown in Figure 3-17 to show when direct-mapped mixed caches are preferred to two-way set-associative mixed caches. Each line is labeled with a miss penalty. The label "M=20" means that a cache miss costs 20 times the time to access a direct-mapped cache. The x-axis shows cache sizes in bytes, while the y-axis shows the access time difference between a two-way set-associative cache and a direct-mapped cache, where the direct-mapped cache has access time of 1.

Given a particular direct-mapped cache and a particular two-way set-associative cache of the same size, this graph can be interpreted as follows: select the point in plane defined by the size of the two caches and their access time difference; if this point is above the expect miss penalty, then the direct-mapped cache is preferred.

The exact location of these lines may be different with miss ratio data from other traces. Nevertheless, this graph shows that, as cache size increases, direct-mapped caches are preferred at smaller access time differences.

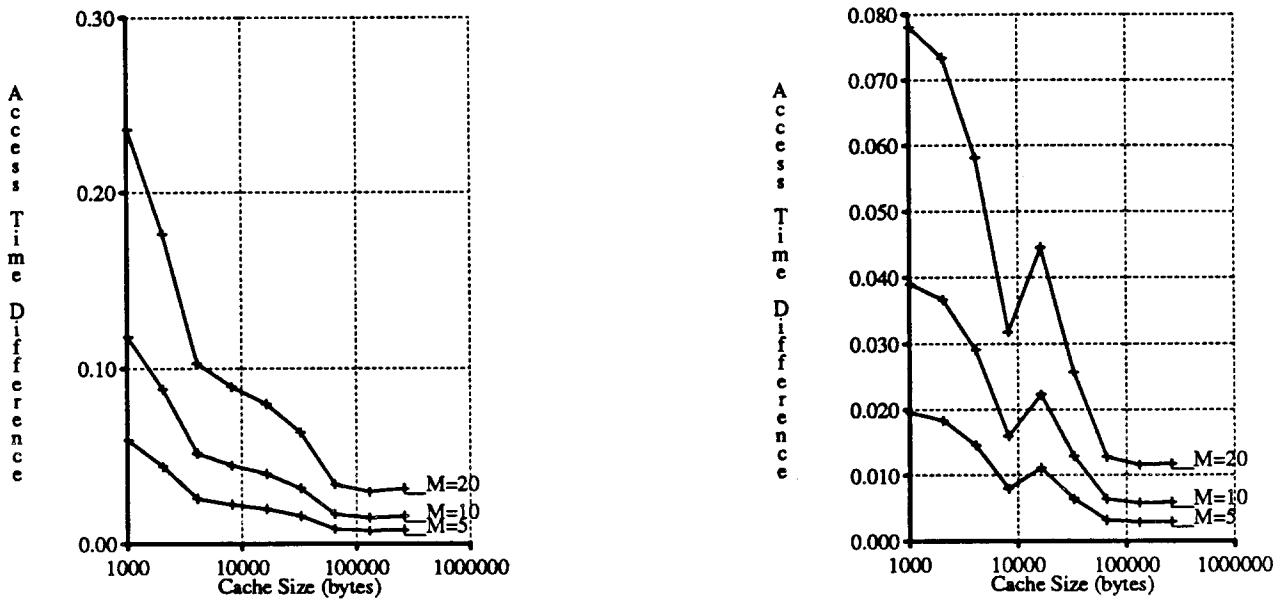


Figure 3-20. More Mixed Cache Crossovers.

This figure compares two-way vs. four-way set-associative mixed caches (left) and four-way vs. eight-way set-associative mixed caches (right) in the same way Figure 3-19 compared direct-mapped vs. two-way set-associative mixed caches. Caches of smaller associativity, two-way (left) and four-way (right), are preferred for points above the appropriate miss penalty line.

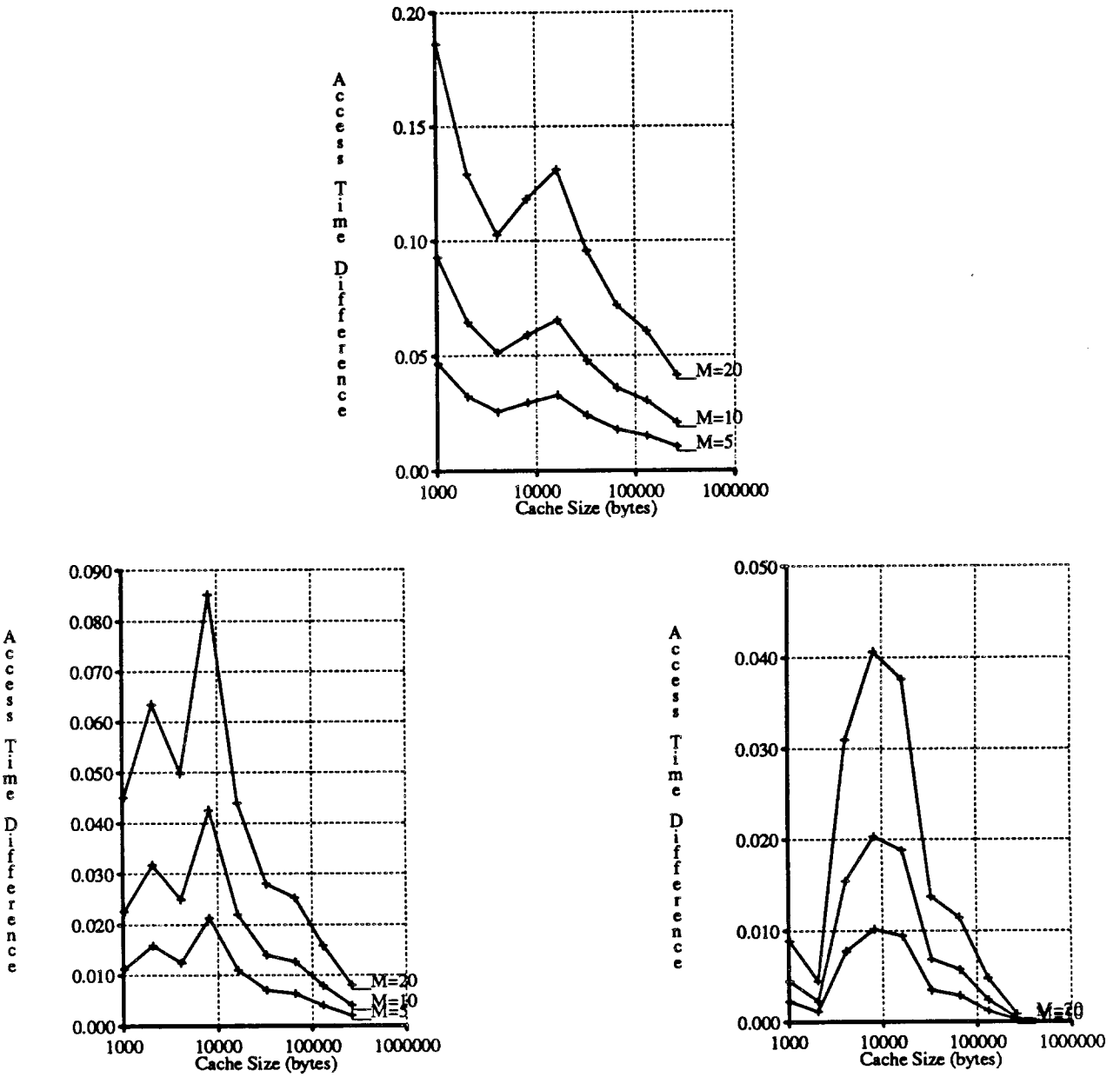


Figure 3-21. Instruction Cache Crossovers.

For direct-mapped vs. 2-way (top), 2- vs.4-way (bottom-left) and 4- vs.8-way (bottom-right).

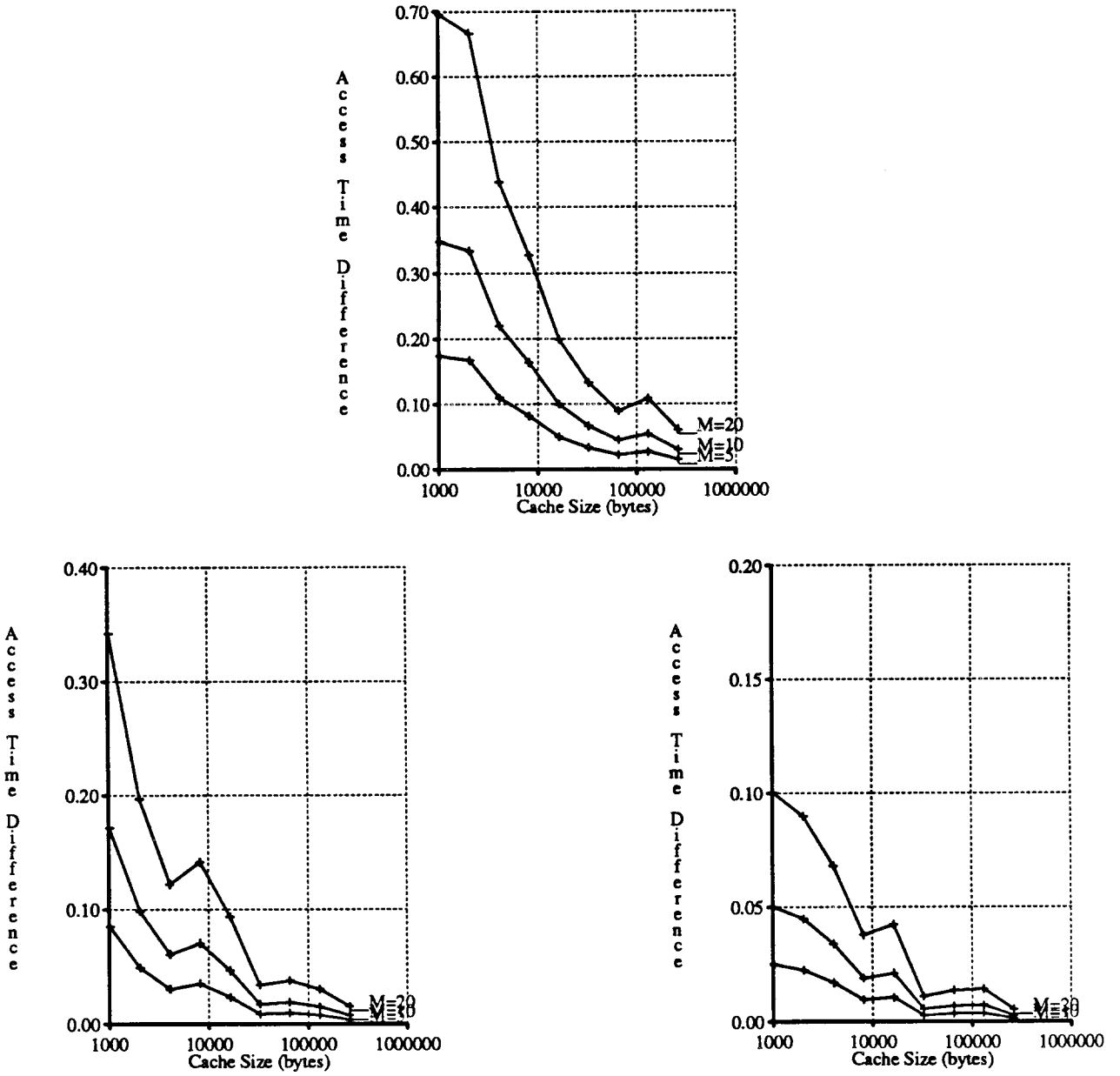


Figure 3-22. Data Cache Crossovers.
For direct-mapped vs. 2-way (top), 2- vs.4-way (bottom-left) and 4- vs.8-way (bottom-right).

3.3.2. A Cache Architecture

This section develops a generic cache architecture to serve as a framework for the cache implementations described in Section 3.5. I concentrate on direct-mapped cache access logic and *set-associativity* logic, because the delay through this logic determines cache access time and directly affects effective access time. *Set-associativity logic* is the additional logic required by a set-associative cache to multiplex alternative data words together.

Assume a single 4G-byte virtual address space of aligned four-byte words, addressed with 32-bit byte-addresses. The memory system, therefore, accepts the 32-bit addresses, discards the two low-order bits, and uses the upper 30 bits to select one of 2^{30} words. Let virtual memory be divided into 4K-byte pages that are mapped to physical page frames by a method that does not determine $t_{cache}(C)$. Address translation does not determine $t_{cache}(C)$, for example, if it is done with a translation buffer in parallel

with cache lookup, as is called "typical" by Smith [Smit82], or if the cache uses virtual address tags and address translation is done only for cache misses, as in SPUR [Wood86a]. Since I am not interested in address translation per se, I assume without loss of generality that the caches in the rest of the paper are accessed with virtual addresses and store virtual address tags. By assuming that the memory system does not support unaligned accesses and that address translation time is hidden, I minimize cache access time and accentuate cache access time differences. Cache access time differences in systems where cache accesses are slowed by unaligned accesses or address translation will be *relatively* smaller.

A direct-mapped cache is simpliest to build, because the cache location of a referenced word is a function of the address of a reference only. A direct-mapped cache lookup requires two parallel actions (see Figure 3-23). One action is to read the word in the cache where the referenced word could reside, and pass it directly back to the CPU. The second action is to read the address tag and state bits for that word to see if the address tag matches and whether the block is valid. A bit is then sent to the CPU indicating a hit if a valid match has occurred, or indicating a miss otherwise. A direct-mapped cache lookup is simpler than a set-associative cache lookup. In a set-associative cache lookup, reading the data and reading the tags are not independent; instead the tags influence the data selected.

An N-way set-associative cache, with N equal to two, four, or eight, is a commonly-used cache organization. An N-way set-associative cache allows any one of the N blocks in the set of a reference to be replaced on a miss. While this flexibility often yields lower miss ratios, it requires that N blocks be searched on each reference. To keep a set-associative cache access time similar to that of a direct-mapped cache, each of the N tags in a set must be read in parallel and compared to the tag of the reference in parallel. This associative lookup and comparison adds significant cost, as measured in chip count and board area, to a set-associative cache relative to a direct-mapped one.

Figure 3-24 shows the basic structure of an N-way set-associative cache. Each bank has the same structure as a direct-mapped cache. In addition, some logic is needed to combine the results of the N banks. This logic, which I call *set-associativity logic*, can take the form of OR-gates and multiplexors (Figure 3-24), or wired-ORs and tri-state buffers (Figure 3-25). The delay through this logic is determined by one of three timing paths: (1) *Match[i]* to *MatchOut*, (2) *Match[i]* through *Select* to *DataOut* and (3) *Data[i]* to *DataOut*.

The distinction between the logic within the N banks and the set-associativity logic is not as clear in many implementations as it is in Figures 3-24 and 3-25. For example, the N comparators and the encoding logic can be combined into a single N-way comparator that directly controls the multiplexor. Nevertheless, a set-associative cache requires some additional circuitry and time beyond that used by a direct-mapped bank to produce *MatchOut* and *DataOut*.

The access time of a set-associative cache is greater than or equal to that of a direct-mapped cache of the same size. It cannot be less since a direct-mapped cache can alway be implemented with the same structure as the set-associative cache, with *Select* determined by address bits rather than by the match logic. Designers of the Titan I, for example, found direct-mapped and four-way set-associative access times equal, since their critical paths ran to *MatchOut* and not to *DataOut*, and the four *Match[i]*'s can be wire-ORed in ECL in zero-time [Niel86]. For this reason and because a set-associative cache has a lower miss ratio, they implemented the four-way set-associative cache.

The access time of a set-associative cache is much greater than that of a direct-mapped cache of the same size when the time to determine which bank hits is much larger than the time to read data from a bank. In a direct-mapped cache, data can be read out of the cache, passed to the CPU, and execution resumed even before *MatchOut* is determined, as long as the CPU can back out of execution begun with incorrect data. This optimistic use of cache data is being used in a research machine [Dion86], where it enables the cache access time and the machine cycle time to be reduced by approximately one-third, and is rumored to be used in some commercially-available workstations†. This technique cannot be used in a straight-forward set-associative cache, because the data returned to the processor is

† The manufacturer declined to confirm these reports, and requested anonymity.

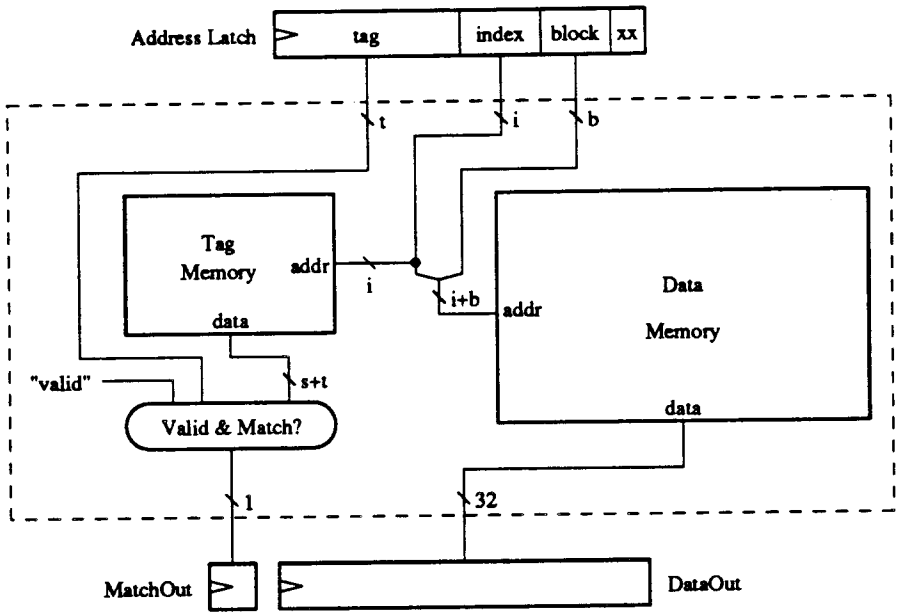


Figure 3-23. A Direct-mapped Cache.

The figure shows the three components of the access logic (not the miss logic) for a direct-mapped cache that selects the set (block frame) of a reference using the fastest method, i.e., bit selection. The first component, the *data memory*, holds all the cached data and instructions. Its size is, by definition, the cache size. Conceptually, it is organized as if it were one word wide and accessed with an address formed by concatenating the index and block fields of the address. If it is implemented as a wider memory, some or all of the bits in the block field shift from addressing memory chips to selecting a word after the memory chip access. A block-wide data memory is often preferred when unaligned memory references are permitted. The second component, the *tag memory*, which holds the address tag and state bits associated with a cached block, is the cache size in blocks deep and is addressed by the index field. The state is often composed of just a valid bit and a dirty bit. The last component, the *match logic*, produces a single bit indicating whether the referenced block is present. This bit is asserted only if the tag read from the tag memory is equal to the tag field of the address and the state read from the tag memory is "valid."

The address of a reference to a direct-mapped cache is divided into several fields, which from least- to most-significant are: 2 bits that are discarded, since I assume aligned word references; $b = \log_2(\textit{block size in words})$ bits of the block (offset); $i = \log_2(\textit{cache size in blocks})$ bits of the index; and $t = 32 - i - b - 2$ bits of the (address) tag.

A direct-mapped cache lookup requires two parallel actions. One action, called *data-lookup*, consists of accessing the data memory with the index and block address fields and passing the word read to *DataOut*. The second action, called *tag-lookup-match*, requires two steps. First the tag memory is accessed using the index address field. Second, the tag and state read from the tag memory is compared by the match logic to assert *MatchOut* when a cache hit is detected.

not selected until after the *Match[i]'s* are computed. A way speed up a set-associative cache on average is to optimistically return data from the most-recently-used block in the selected set [Chan87]. I discuss this approach in Section 3.3.4.

Finally, the access time of a set-associative cache can be a little larger than that of a direct-mapped cache of the same size if the set-associative access is slower only by the delay through the set-associativity logic. The access time difference is exactly equal to the delay through the set-associativity logic if the delay through the C direct-mapped cache is exactly equal to the delay through N parallel direct-mapped banks of C/N blocks. My examination of memory chips suggests that the two delays are often comparable, particularly since the direct-mapped caches can use larger memory chips that are not much slower and whose use reduces the load on address drivers. I also found this relationship to hold for bank delays in the SPUR cache.

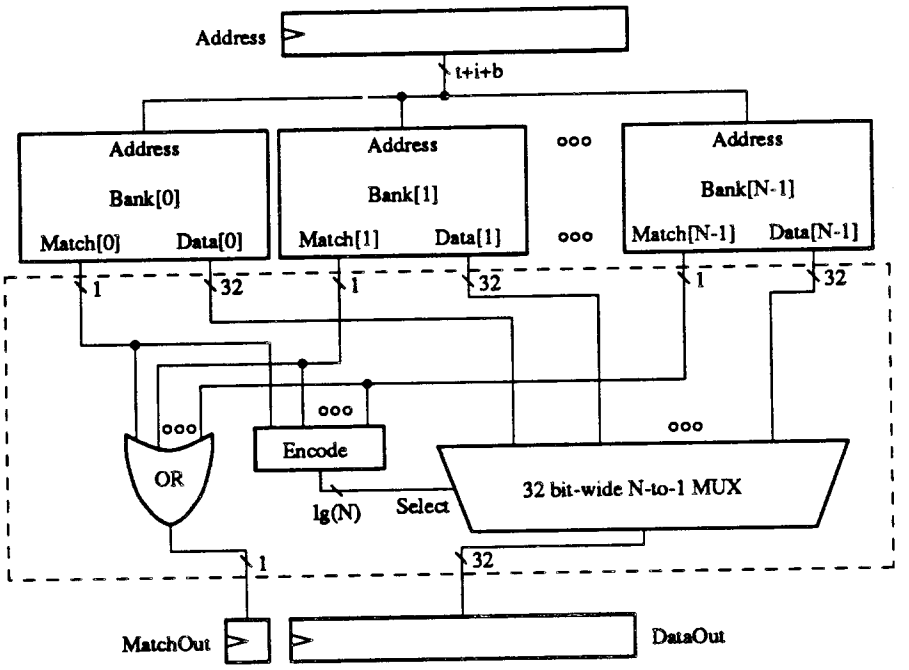


Figure 3-24. A Set-Associative Cache.

This figure shows cache access logic for an N-way set-associative cache. This logic consists of N direct-mapped banks and logic to combine the N results into a single match signal and data word.

Each bank in an N-way set-associative cache of C blocks can be thought of and implemented as a direct-mapped cache of C/N blocks. On a reference, the address is passed to all the direct-mapped banks. In parallel, each bank selects a block, sends 32-bits of data to *Data[i]*, and computes *Match[i]*, which is asserted on valid tag matches. The *set* of a reference consists of the N blocks selected by the N banks.

After the N direct-mapped caches compute *Match[i]*'s and *Data[i]*'s, the additional logic shown in the dashed box is necessary to produce a single *MatchOut* signal and *DataOut* word. *MatchOut*, which is asserted on a cache hit is the logical OR of the N *Match[i]* signals. One way to compute it is with a single N-input OR gate. *Select*, an internal signal, is the number of the bank that matched and can be any value if none matched. *Select* can be computed with an N-bit encoder or with a single level of $\log_2(N)$ $N/2$-input OR gates. *DataOut* must be driven to the value of the bank that matched and can be any value if none matched. One way to compute *DataOut* is with a 32-bit-wide N-to-1 multiplexor (MUX).

In the next section, I compare caches implemented in AS TTL, ECL 100K and custom CMOS to illustrate that implementation factors should be considered when comparing caches with effective access time. For the reasons give above, I will assume that set-associative caches are slower than direct-mapped ones by exactly the delay through set-associativity logic. More details regarding cache implementations can be found in Section 3.5.

3.3.3. Comparing Effective Access Times

Here I compare the effective access times direct-mapped and set-associative caches implemented in AS TTL, ECL 100K and custom CMOS to show that effective access time comparisons which ignore access time differences can be misleading. In particular, I find that direct-mapped caches of large, practical sizes can yield similar or lower effective access times than those of set-associative caches of the same size.

Let the *5-percent size* be the minimum cache size for which the effective access time of a direct-mapped cache is similar to that of set-associative caches of the same size. I arbitrarily define "similar" to be "within 5 percent," because a 5 percent difference in effective access time translates into no more than a 5 percent difference in workload execution time, since computers do not spend all their time

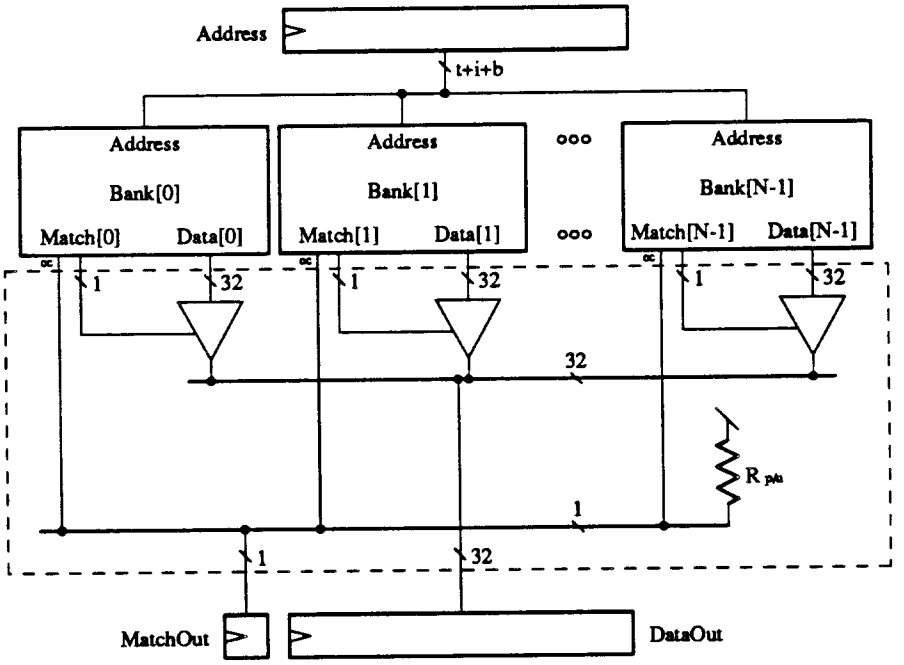


Figure 3-25. An Alternative Set-Associative Cache.

This figure shows cache hit logic for an N-way set-associative cache using alternative implementation style. The 32-bit wide N-to-1 multiplexor and select logic has been replaced with N 32-bit-wide tri-state buffers. This alternative does not have to encode *Select*. Instead, each Bank[i] independently enables its tri-state buffer to drive *Data[i]* to *DataOut* if its *Match[i]* is asserted. At most one bank drives *DataOut* since more than one bank cannot match. If no banks match, *DataOut* is undefined. The N *Data[i]* 's are connected to each other and *DataOut* with one 32-bit bus. To first order and for small values of N, e.g., $N \leq 8$, the delay to *DataOut* with this design is independent of N.

Alternative logic using wired-OR is also illustrated here for computing *MatchOut*. Each *Match[i]* is computed twice (in parallel) by duplicating the final OR-gate in the match logic (not shown). One copy drives the tri-state enable or multiplexor select, and the other is wired to the other *Match[i]* 's. In TTL, the *Match[i]* 's that are wire-ORed together must be produced with OR-gates using open-collector outputs ("oc"). In ECL, any outputs can be wire-ORed. Two copies of each *Match[i]* are necessary so that the wire-ORing does not affect which data is selected.

waiting for memory. Similarly, let the *crossover size* be the minimum cache size for which the effective access time of a direct-mapped cache is less than that of set-associative caches of the same size. The 5-percent size is interesting when direct-mapped caches "cost" less than set-associative caches (in design or debug time, chip area, chip count, power consumption, etc.) making similar performance adequate. The crossover size is interesting when direct-mapped and set-associative caches cost the same. Since both cases can occur, I present both the 5-percent and crossover sizes for various technologies, cache types and miss penalties.

3.3.3.1. TTL Caches

Effective access times for numerous TTL caches are presented in Figures 3-26 and 3-27 and in Tables 3-16 through 3-19. Results use warm-start miss ratios from trace average *2nd500k*, the access time assumptions in Table 3-15, "slow" and "fast" miss penalties. The slow miss penalty, 12 cycles plus one cycle per word, is the same as the miss penalty in SPUR, which services misses with an extended version of a standard 32-bit backplane bus. The fast miss penalty, 6 cycles plus one cycle per doubleword, corresponds to using "fast" miss logic and a special-purpose 64-bit memory interconnect. Caution should be used when comparing caches of radically different sizes, because cache access times are affected by large changes in cache size.

Degree of Associativity	Cache Access Time (ns)	Cache Access Time % increase from A=1
1	100.0	0.0
2	109.0	9.0
4	109.0	9.0
8	109.0	9.0

Table 3-15. TTL Cache Access Times.

This table lists TTL cache access times, based on the assumptions of Section 3.5. The direct-mapped cache has an access time of 100 ns with a critical path to *Match*. The critical path though the set-associativity logic is from *Match* to *DataOut*. The length of this path is minimized by using tri-state drivers rather than explicit multiplexors. Caches with associativities two, four and eight have the same access times, 109 ns, because, to first-order, the delay through tri-state buffers is not affected by how many of them are operating in parallel.

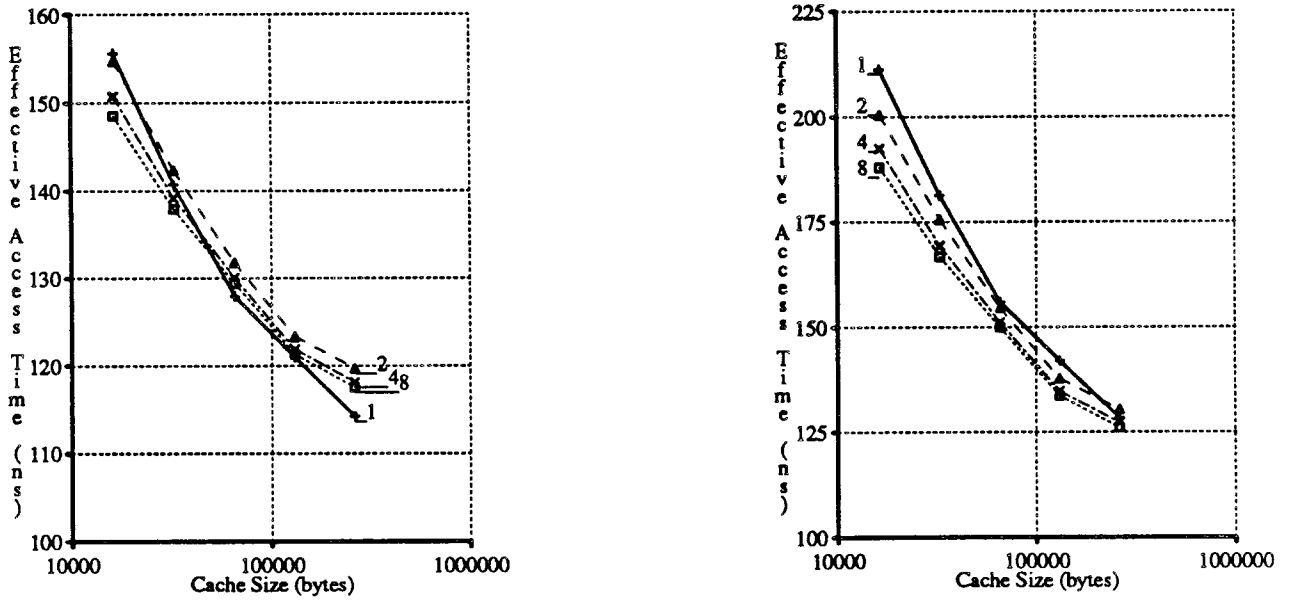


Figure 3-26. Effective Access Times for TTL Mixed Caches.

This figure shows effective access times for TTL mixed caches with 32-byte blocks. The left plot shows effective access times with a "fast" miss penalty of ten 100 ns cycles; the right plot shows times with a "slow" twenty-cycle miss penalty. Miss ratios are based on warm-start simulations with trace average *2nd500k*, while cache access times on are based on TTL implementation assumptions from Section 3.5. This data is displayed in tabular form in Tables 3-16 and 3-17.

For the fast miss penalty (left), direct-mapped cache effective access times are within five percent or better than those of set-associative caches for all sizes shown. For the slow miss penalty (right), direct-mapped caches do relatively worse, since they miss more often than do set-associative caches. Nevertheless, direct-mapped effective access times are within 5 percent of a *two*-way set-associative times for all sizes shown.

Perhaps the most striking feature of the data is how little changes in associativity affect the effective access times of these caches. For example at a given size, most cache configurations in Figure 3-26 have effective access times that differ by less than five percent. I highlight this for each cache size in Tables 3-16 through 3-19 by displaying effective access times in **bold** that are within five percent of the minimum time (indicated with an asterisk). Symbolically, a ratio of relative effective access time changes to relative miss ratio changes is:

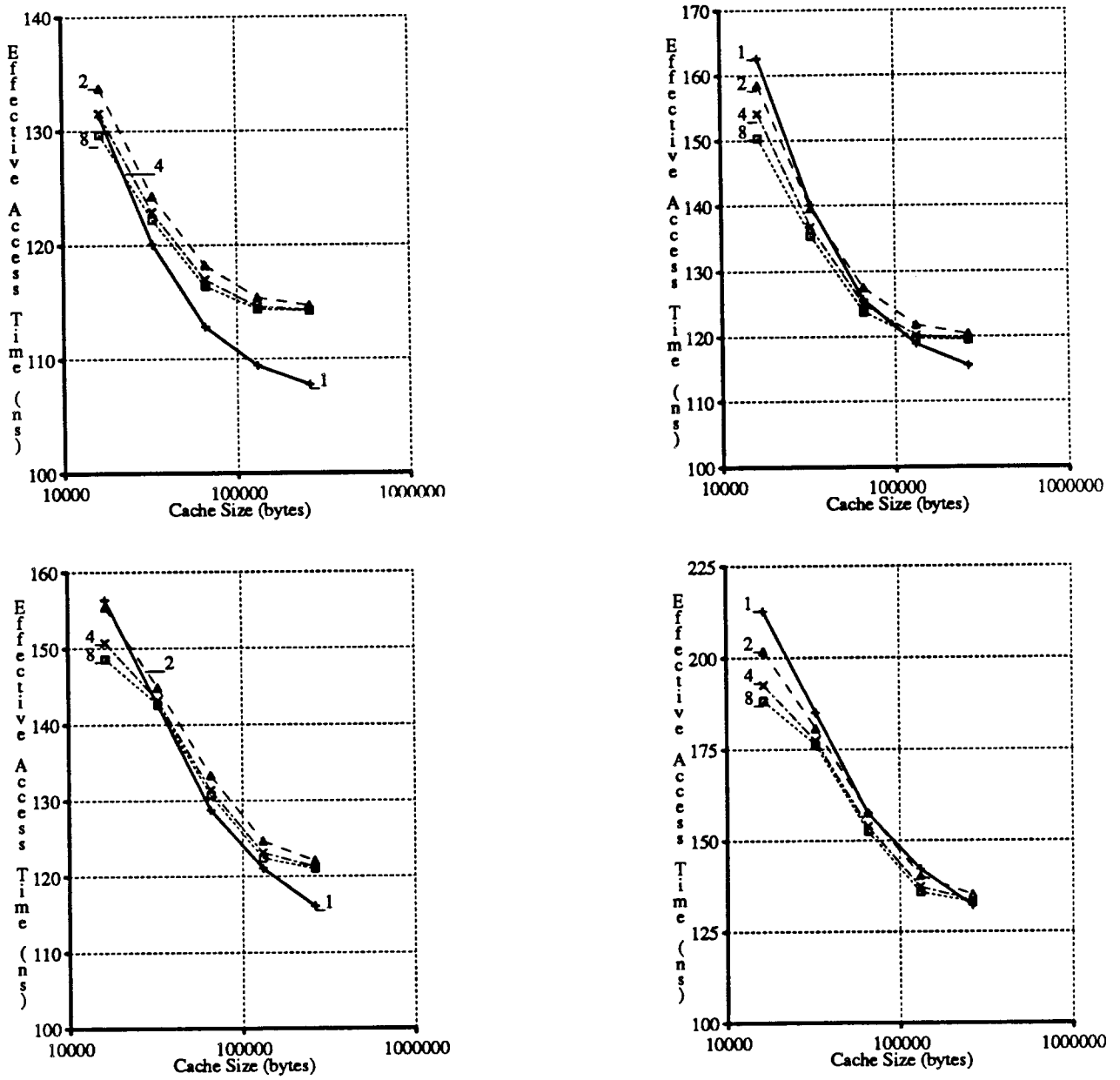


Figure 3-27. More TTL Effective Access Times.

Fast misses (left), slow misses (right), I-caches (top) and D-caches (bottom).

$$\frac{\Delta t_{eff} / t_{eff}}{\Delta m / m} = \left[\frac{m}{t_{eff}}\right]\left[\frac{\Delta t_{cache} + \Delta m^* t_{memory}}{\Delta m}\right]$$

Reducing associativity from 8-way to direct-mapped in a 128K-byte mixed cache with the fast penalty, for example, increases the miss ratio by 71 percent (see Table 3-4), but decreases effective access time by 0.25 percent:

$$\frac{\Delta t_{eff} / t_{eff}}{\Delta m / m} = \left[\frac{0.0123}{1.210}\right]\left[\frac{-0.09 + 0.0087^* 10}{0.0087}\right] = \frac{-0.25\%}{71\%}$$

The effective access time change here is so small, because the both caches are large and almost always hit, i.e., t_{cache} is large relative to $m^* t_{memory}$. For small caches, on the other hand, relative changes in miss ratio are important since $m^* t_{memory}$ is large compared to t_{cache}.

Effective Access Times (ns) for Mixed Caches in AS TTL with a miss penalty of 600.0 + 12.5*block size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes)		
		16	32	64
16384	1-way	157.3	**155.6**	165.6
16384	2-way	156.5	**154.7**	160.6
16384	4-way	**152.4**	**150.8**	**155.7**
16384	8-way	**149.8**	**148.5***	**153.5**
32768	1-way	**141.6**	**140.7**	147.4
32768	2-way	**143.1**	**142.3**	145.9
32768	4-way	**140.1**	**139.2**	**141.6**
32768	8-way	**138.9**	**137.9***	**140.1**
65536	1-way	**128.8**	**128.0***	**132.2**
65536	2-way	**132.2**	**131.8**	134.5
65536	4-way	**130.1**	**130.1**	**132.8**
65536	8-way	**129.1**	**129.5**	**132.1**
131072	1-way	**121.8**	**121.0***	**124.0**
131072	2-way	**124.5**	**123.4**	**124.8**
131072	4-way	**122.7**	**121.9**	**123.5**
131072	8-way	**121.8**	**121.3**	**123.0**
262144	1-way	**115.7**	**114.2***	**115.1**
262144	2-way	121.4	**119.7**	120.0
262144	4-way	120.0	**118.1**	**118.0**
262144	8-way	**119.5**	**117.5**	**117.3**

Table 3-16. Times for Fast TTL Mixed Caches.

This table shows effective access times for TTL mixed caches with 16-, 32- and 64-byte blocks, assuming a "fast" miss penalty. Miss ratios are based on warm-start simulations with trace average *2nd500k*, while cache access times on are based on TTL implementation assumptions from Section 3.5. The smallest effective access time at each cache size is indicated with an asterisk; effective access times within five percent of this time are display in **bold.**

A second trend present in the data is that the effective access times of direct-mapped caches improve relative to those of set-associative caches as cache size increases, confirming the expectation of Section 3.3.1. This trend occurs since the term that gives direct-mapped caches worse performance than set-associative caches, $\Delta m * t_{memory}$, decreases as caches get larger (see Figure 3-17), while the term that gives direct-mapped caches better performance, Δt_{cache}, remains constant. The term $\Delta m * t_{memory}$ gets smaller, because Δm gets smaller. One way to explain diminishing values of Δm is by factoring it into the product of diminishing miss ratios and relatively-constant miss ratios spreads. Alternately, this trend can be explained by noting that direct-mapped effective access times are proceeding asymptotically toward 100 ns, whereas set-associative times are approaching 109 ns.

A third trend is that direct-mapped caches do relatively worse when t_{memory} is increased, as was anticipated is Section 3.3.1 where Figure 3-19, for example, showed that increasing the miss penalty increases the access time penalty necessary for crossover. This trend can also be explained by examining the above equations or simply by observing that direct-mapped caches should be more adversely affected by an increased miss penalty, since they miss more often that do set-associative caches.

A fourth trend is that direct-mapped caches do relatively better for instruction caches as compared to mixed and data caches. This can be attributed to generally lower instruction cache miss ratios and miss ratio spreads. As discussed earlier, instruction caches have lower miss ratio spreads, because instruction references exhibit more spatial locality and lower set-conflicts. Figure 3-21 also show that the crossover points for instruction cache occur at a much small access time differences than do the crossover points for mixed and data caches (see Figures 3-19 and 3-22).

Effective Access Times (ns) for Mixed Caches in AS TTL with a miss penalty of 1200.0 + 25.0*block_size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
16384	1-way	214.6	211.2	231.3
16384	2-way	204.0	200.5	212.2
16384	4-way	**195.8**	**192.5**	202.4
16384	8-way	**190.6**	**188.1***	198.0
32768	1-way	183.3	181.4	194.8
32768	2-way	177.2	175.7	182.7
32768	4-way	**171.1**	**169.3**	**174.1**
32768	8-way	**168.8**	**166.8***	**171.3**
65536	1-way	157.6	**156.0**	164.3
65536	2-way	**155.5**	**154.6**	160.0
65536	4-way	**151.1**	**151.2**	**156.6**
65536	8-way	**149.3***	**149.9**	**155.1**
131072	1-way	143.6	141.9	148.0
131072	2-way	**140.0**	**137.7**	140.5
131072	4-way	**136.3**	**134.7**	**138.0**
131072	8-way	**134.6**	**133.6***	**137.0**
262144	1-way	**131.4**	**128.4**	**130.3**
262144	2-way	133.8	**130.4**	**130.9**
262144	4-way	**131.0**	**127.2**	**127.1**
262144	8-way	**130.1**	**126.1**	**125.7***

Table 3-17. Times for Slow TTL Mixed Caches.

This table shows effective access times for TTL mixed caches, assuming a "slow" miss penalty. See the caption of Table 3-16 for more details.

Finally, Table 3-20 shows the minimum cache sizes for which direct-mapped caches have similar (*5-percent size*) or better (*crossover size*) effective access times than two-way, four-way and eight-way set-associative caches of the same size. The values of these sizes are sensitive to small changes in assumptions, which can move the exact place where these almost parallel lines cross. Nevertheless, results show that direct-mapped caches have similar or better effective access times at cache sizes now considered practical.

Effective Access Times (ns) for Instruction Caches in AS TTL with a miss penalty of 600.0 + 12.5*block_size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes)		
		16	32	64
16384	1-way	137.0	**131.3**	**130.9**
16384	2-way	137.9	**133.8**	**133.9**
16384	4-way	**135.3**	**131.5**	**131.7**
16384	8-way	**133.6**	**129.7**	**129.6***
32768	1-way	**123.6**	**120.1**	**120.0***
32768	2-way	126.9	**124.3**	**124.2**
32768	4-way	**125.1**	**122.9**	**122.8**
32768	8-way	**124.2**	**122.2**	**122.2**
65536	1-way	**115.4**	**112.8**	**112.6***
65536	2-way	120.2	**118.2**	**118.1**
65536	4-way	118.5	**117.0**	**116.9**
65536	8-way	**117.8**	**116.4**	**116.5**
131072	1-way	**111.5**	**109.4**	**109.1***
131072	2-way	117.2	115.4	114.9
131072	4-way	116.3	114.6	**114.1**
131072	8-way	116.1	**114.4**	**113.7**
262144	1-way	**109.7**	**107.8**	**107.3***
262144	2-way	116.4	114.7	114.1
262144	4-way	116.1	114.3	113.6
262144	8-way	116.0	114.2	113.5

Effective Access Times (ns) for Instruction Caches in AS TTL with a miss penalty of 1200.0 + 25.0*block_size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes)		
		16	32	64
16384	1-way	174.1	162.6	161.9
16384	2-way	166.8	158.5	158.7
16384	4-way	161.7	**154.1**	**154.4**
16384	8-way	158.2	**150.3**	**150.1***
32768	1-way	147.2	**140.1**	**140.1**
32768	2-way	144.8	**139.6**	**139.3**
32768	4-way	**141.3**	**136.8**	**136.6**
32768	8-way	**139.4**	**135.4**	**135.3***
65536	1-way	130.7	**125.7**	**125.2**
65536	2-way	131.3	**127.5**	**127.2**
65536	4-way	**128.1**	**124.9**	**124.8**
65536	8-way	**126.5**	**123.8***	**124.0**
131072	1-way	**122.9**	**118.8**	**118.2***
131072	2-way	125.3	**121.8**	**120.9**
131072	4-way	**123.5**	**120.2**	**119.2**
131072	8-way	**123.2**	**119.7**	**118.5**
262144	1-way	**119.4**	**115.5**	**114.5***
262144	2-way	123.9	120.4	**119.1**
262144	4-way	123.1	**119.6**	**118.1**
262144	8-way	123.1	**119.5**	**118.0**

Table 3-18. Times for TTL Instruction Caches.

Effective Access Times (ns) for Data Caches in AS TTL with a miss penalty of 600.0 + 12.5*block size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes)		
		16	32	64
16384	1-way	**150.8**	156.4	174.7
16384	2-way	**151.4**	155.4	166.0
16384	4-way	**148.7**	**150.8**	159.6
16384	8-way	**147.7***	**148.6**	155.8
32768	1-way	**138.1**	**142.5**	154.9
32768	2-way	**140.8**	144.9	152.4
32768	4-way	**138.6**	**143.2**	148.5
32768	8-way	**137.6***	**142.6**	146.9
65536	1-way	**127.3***	**128.8**	136.3
65536	2-way	**131.4**	**133.3**	139.4
65536	4-way	**129.3**	**131.4**	137.9
65536	8-way	**128.3**	**130.7**	137.8
131072	1-way	**121.3**	**121.1***	**125.8**
131072	2-way	**125.9**	**124.7**	**126.8**
131072	4-way	**124.6**	**123.2**	**125.2**
131072	8-way	**124.1**	**122.4**	**124.4**
262144	1-way	**117.7**	**116.1***	**117.5**
262144	2-way	124.3	122.1	122.4
262144	4-way	123.8	**121.3**	**121.1**
262144	8-way	123.6	**121.0**	**120.5**

Effective Access Times (ns) for Data Caches in AS TTL with a miss penalty of 1200.0 + 25.0*block size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes)		
		16	32	64
16384	1-way	201.7	212.8	249.4
16384	2-way	**193.8**	201.9	222.9
16384	4-way	**188.4**	**192.5**	210.2
16384	8-way	**186.4***	**188.3**	202.5
32768	1-way	176.2	185.0	209.8
32768	2-way	**172.6**	180.8	195.8
32768	4-way	**168.1**	177.3	188.0
32768	8-way	**166.2***	176.3	184.8
65536	1-way	**154.5**	157.5	172.6
65536	2-way	**153.8**	157.6	169.9
65536	4-way	**149.6**	**153.8**	166.7
65536	8-way	**147.5***	**152.5**	166.7
131072	1-way	**142.7**	**142.2**	151.6
131072	2-way	142.9	**140.3**	144.7
131072	4-way	**140.2**	**137.3**	**141.5**
131072	8-way	**139.2**	**135.9***	**139.8**
262144	1-way	**135.3**	**132.2**	**134.9**
262144	2-way	139.7	**135.1**	**135.9**
262144	4-way	**138.6**	**133.6**	**133.1**
262144	8-way	**138.2**	**133.1**	**132.0***

Table 3-19. Times for TTL Data Caches.

Minimum TTL Cache Size where Direct-Mapped Caches have Similar or Smaller Effective Access Times			
Cache Type	Miss Time	5-Percent Size	Crossover Size
Mixed	fast	16K	64K
Instruction	fast	≤1K	32K
Data	fast	16K	64K
Mixed	slow	64K	>256K
Instruction	slow	32K	128K
Data	slow	64K	256K

Table 3-20. Crossover for TTL Caches.

This table shows *5-percent* and *crossover* sizes for TTL caches. The 5-percent size is the minimum cache size where the effective access time of a direct-mapped TTL cache is within five percent or is smaller than the smallest effective access time for that cache size. The crossover size is the minimum cache size where the effective access time of a direct-mapped TTL cache is smaller than the effective access times for two-way, four-way and eight-way set-associative caches of that cache size.

Results for the fast miss time show that the direct-mapped and set-associative cache effective access times are within five percent of all cache sizes shown. Results with the slow miss time show that direct-mapped cache effective access times are similar to those of set-associative caches for cache sizes greater than 32K bytes.

3.3.3.2. ECL Caches

Now I consider implementing caches in ECL to examine whether the conclusions reached with TTL caches generalize. The ECL access time assumptions are discussed in Table 3-21 and Section 3.5. I report results for smaller caches, however (2K to 32K bytes instead of 16K to 256K bytes), since my experience suggests that, at any point in time, ECL caches tend to be smaller than TTL caches, because ECL logic tends to consume more power than TTL logic and ECL RAMs tend to be less dense than CMOS RAMs. I assume the same miss penalties measured in numbers of cycles, but will assume a 30.0 ns rather than a 100 ns cycle time. Consequently, I expect that a faster, more expensive ECL cache will be interfaced to a faster, more expensive memory system. The ECL Titan-I cache miss penalty of 13 cycles for a 16-byte block, for example, is between my fast and slow miss penalties [Joup86]. ECL effective access times are displayed in Figures 3-26 and 3-27 and Tables 3-22 through 3-25.

Degree of Associativity	Cache Access Time (ns)	Cache Access Time % increase from A=1
1	30.0	0.0
2	33.5	11.7
4	34.4	14.7
8	34.4	14.7

Table 3-21. ECL Cache Access Times.

This table lists ECL cache access times, based on the assumptions of Section 3.5. The direct-mapped cache has an access time of 30 ns with a critical path to *Match*. The critical path though the set-associativity logic is from *Match* to *DataOut*. I implement this logic with multiplexors, because using driver chips adds too many chips to the cache implementation. In a four-way set-associative cache, for example, using drivers adds 32 chips, while using multiplexors adds 12. If board area is available for the 20 extra chips, using multiplexors is still preferred, since the 20 chip positions can be filled with memory chips to significantly increase cache size.

The TTL results do generalize to ECL caches, with two minor differences. First, 5-percent sizes and crossover sizes (see Table 3-26) are smaller for ECL than for TTL, because my assumptions show adding set-associativity logic to an ECL cache causes a larger relative increase in cache access time than adding such logic to a TTL cache. Set-associativity logic slows ECL caches by 12 to 15 percent,

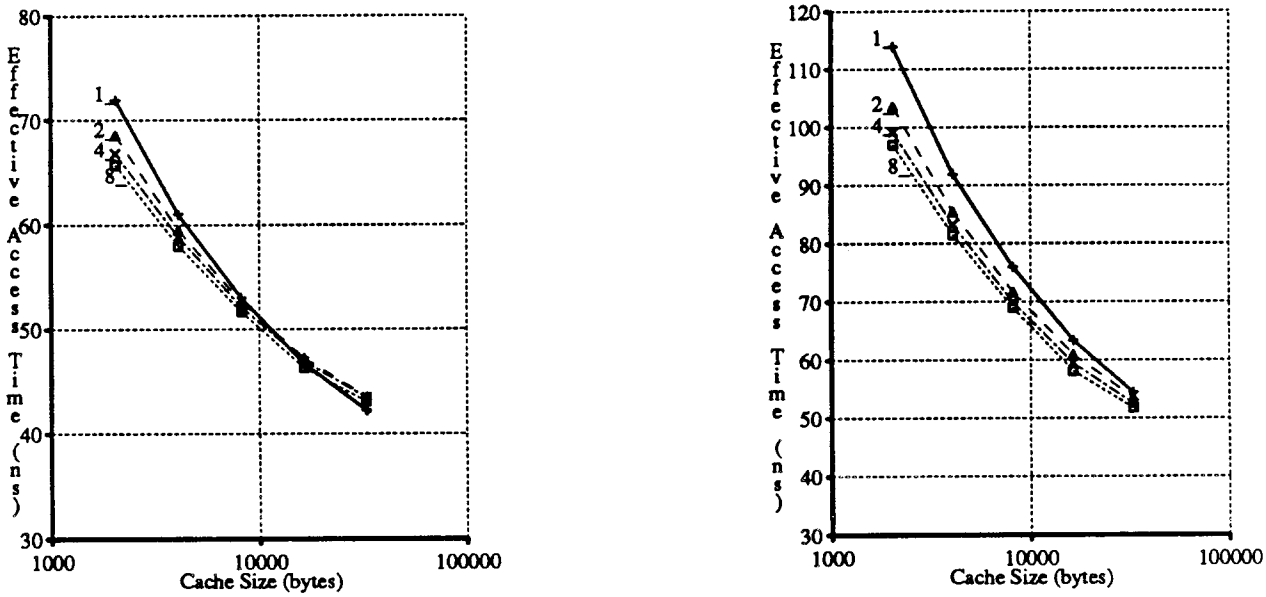


Figure 3-28. Effective Access Times for ECL Mixed Caches.

This figure shows effective access times for ECL mixed caches with 32-byte blocks. Cache access times of 30.0, 33.5, 34.4 and 34.4 ns for direct-mapped, two-, four- and eight-way set-associative caches are based on ECL implementation assumptions from Section 3.5; miss penalties are the same as TTL miss penalties, measured in 30.0 ns rather than 100 ms cycles. This data is displayed in tabular form in Tables 3-22 and 3-23.

Direct-mapped effective access times are similar to set-associative ones with a fast miss penalty (left), and slightly worse with a slow miss penalty (right).

and TLL caches by only 9 percent. The differences in 5-percent and crossover size underscores the sensitivity of these sizes to small changes in implementation assumptions, rather than conclusively showing that ECL and direct-mapped caches are well-matched. Second, if set-associativity logic added the same relative delay to ECL and TTL cache times, for example 9 percent, then ECL direct-mapped caches would be less attractive than TTL direct-mapped caches, because ECL caches are smaller than TTL caches, and, as argued earlier, the preference toward direct-mapped caches increases with cache size.

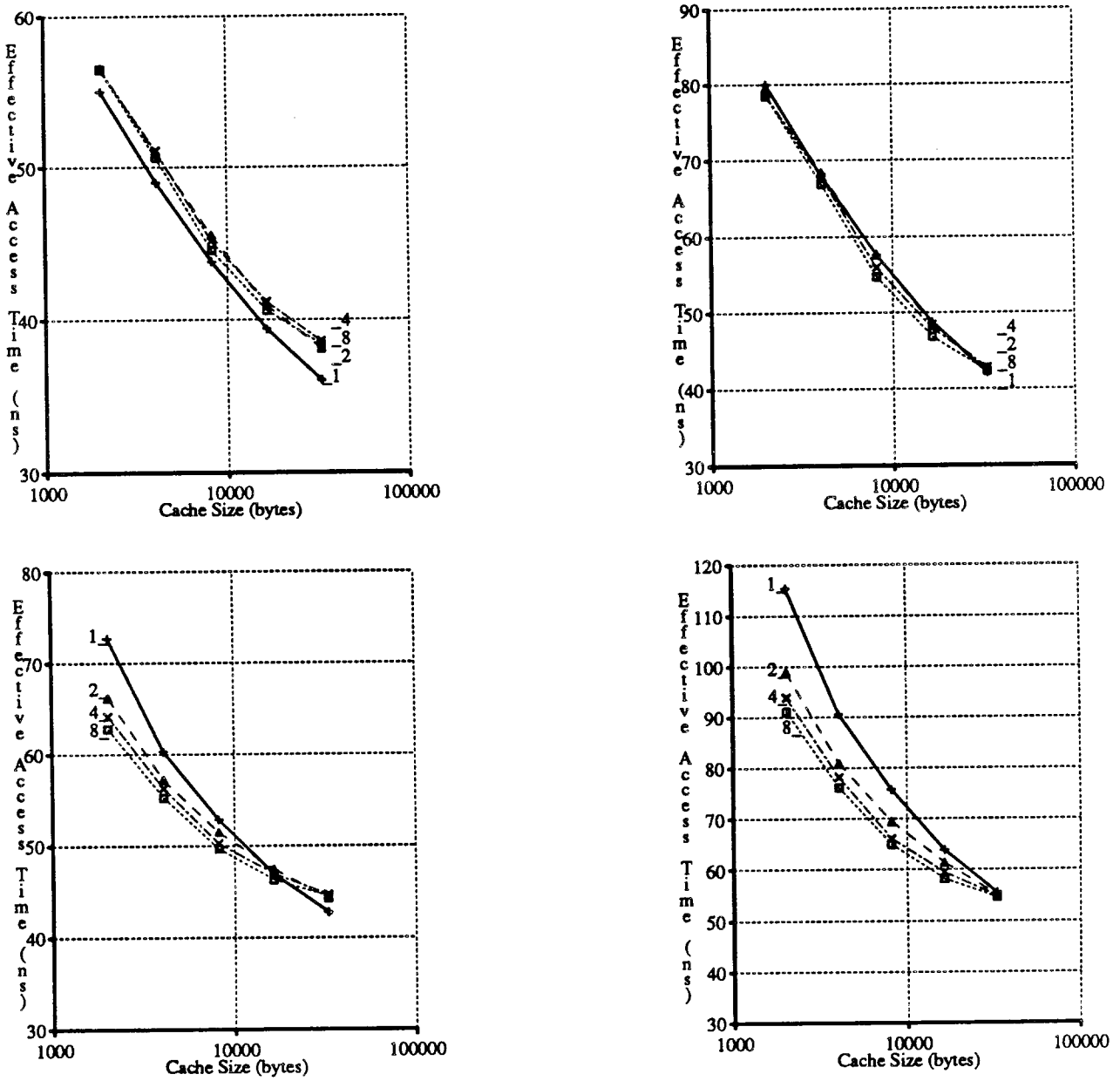


Figure 3-29. More ECL Effective Access Times.

Fast misses (left), slow misses (right), I-caches (top) and D-caches (bottom).

Effective Access Times (ns) for Mixed Caches in ECL 100K with a miss penalty of 180.0 + 3.75*block_size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
2048	1-way	71.1	71.9	83.6
2048	2-way	69.3	**68.6**	76.2
2048	4-way	**67.8**	**66.8**	73.0
2048	8-way	**66.9**	**65.7***	71.4
4096	1-way	61.1	61.0	68.6
4096	2-way	**60.4**	**59.5**	63.9
4096	4-way	**59.9**	**58.8**	62.4
4096	8-way	**58.9**	**58.0***	61.3
8192	1-way	**53.6**	**53.0**	58.0
8192	2-way	**53.3**	**52.6**	55.4
8192	4-way	**52.8**	**52.1**	54.6
8192	8-way	**52.5**	**51.7***	**53.9**
16384	1-way	**47.2**	**46.7**	49.7
16384	2-way	**47.8**	**47.2**	49.0
16384	4-way	**47.4**	**46.9**	**48.4**
16384	8-way	**46.6**	**46.3***	**47.8**
32768	1-way	**42.5**	**42.2***	**44.2**
32768	2-way	**43.7**	**43.5**	44.6
32768	4-way	**43.7**	**43.5**	**44.2**
32768	8-way	**43.4**	**43.1**	**43.7**

Table 3-22. Times for Fast ECL Mixed Caches.

This table shows effective access times for ECL mixed caches with 16-, 32- and 64-byte blocks, assuming a "fast" miss penalty. Miss ratios are based on warm-start simulations with trace average *2nd500k*, while cache access times on are based on ECL implementation assumptions from Section 3.5. The smallest effective access time at each cache size is indicated with an asterisk; effective access times within five percent of this time are display in **bold.**

Effective Access Times (ns) for Mixed Caches in ECL 100K with a miss penalty of 360.0 + 7.5*block_size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
2048	1-way	112.3	113.9	137.2
2048	2-way	105.1	103.6	118.8
2048	4-way	**101.3**	**99.2**	111.6
2048	8-way	**99.4**	**97.0***	108.4
4096	1-way	92.3	92.0	107.3
4096	2-way	87.4	**85.5**	94.2
4096	4-way	**85.5**	**83.3**	90.4
4096	8-way	**83.4**	**81.6***	88.2
8192	1-way	77.2	76.0	85.9
8192	2-way	73.2	**71.7**	77.3
8192	4-way	**71.3**	**69.9**	74.8
8192	8-way	**70.6**	**68.9***	73.4
16384	1-way	64.4	63.4	69.4
16384	2-way	62.0	**60.9**	64.5
16384	4-way	**60.5**	**59.5**	62.4
16384	8-way	**58.9**	**58.1***	61.1
32768	1-way	55.0	54.4	58.4
32768	2-way	**54.0**	**53.5**	55.6
32768	4-way	**53.0**	**52.5**	**53.9**
32768	8-way	**52.3**	**51.7***	**53.1**

Table 3-23. Times for Slow ECL Mixed Caches.

This table shows effective access times for ECL mixed caches, assuming a "slow" miss penalty. See the caption of Table 3-22 for more details.

Effective Access Times (ns) for Instruction Caches in ECL 100K with a miss penalty of 180.0 + 3.75*block_size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
2048	1-way	59.4	**55.0***	**55.5**
2048	2-way	61.2	**56.6**	**56.3**
2048	4-way	61.0	**56.5**	**56.4**
2048	8-way	60.8	**56.4**	**56.5**
4096	1-way	52.5	**49.0***	**49.4**
4096	2-way	54.1	**51.0**	**50.8**
4096	4-way	54.0	**51.1**	**51.0**
4096	8-way	53.4	**50.6**	**50.7**
8192	1-way	46.3	**43.8***	**43.9**
8192	2-way	47.4	**45.6**	**45.7**
8192	4-way	46.6	**45.2**	**45.7**
8192	8-way	**46.0**	**44.6**	**45.3**
16384	1-way	**41.1**	**39.4**	**39.3***
16384	2-way	42.2	**40.9**	**41.0**
16384	4-way	42.3	**41.2**	**41.2**
16384	8-way	41.8	**40.6**	**40.6**
32768	1-way	**37.1**	**36.0**	**36.0***
32768	2-way	38.9	38.1	38.0
32768	4-way	39.2	38.6	38.5
32768	8-way	39.0	38.4	38.3

Effective Access Times (ns) for Instruction Caches in ECL 100K with a miss penalty of 360.0 + 7.5*block_size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
2048	1-way	88.8	**80.0**	**81.0**
2048	2-way	89.0	**79.6**	**79.1**
2048	4-way	87.5	**78.6**	**78.5***
2048	8-way	87.1	**78.5**	**78.6**
4096	1-way	75.0	**68.0**	**68.8**
4096	2-way	74.6	**68.4**	**68.0**
4096	4-way	73.7	**67.8**	**67.7**
4096	8-way	72.4	**66.9***	**66.9**
8192	1-way	62.7	**57.7**	**57.9**
8192	2-way	61.2	**57.6**	**58.0**
8192	4-way	58.9	**56.0**	**56.9**
8192	8-way	57.6	**54.7***	**56.1**
16384	1-way	52.2	**48.8**	**48.6**
16384	2-way	50.8	**48.4**	**48.4**
16384	4-way	50.2	**47.9**	**48.0**
16384	8-way	49.2	**46.8**	**46.7***
32768	1-way	44.2	**42.0**	**42.0***
32768	2-way	44.2	**42.7**	**42.6**
32768	4-way	**44.1**	**42.7**	**42.7**
32768	8-way	**43.5**	**42.3**	**42.3**

Table 3-24. Times for ECL Instruction Caches.

Effective Access Times (ns) for Data Caches in ECL 100K with a miss penalty of 180.0 + 3.75*block_size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
2048	1-way	66.4	72.7	89.7
2048	2-way	**61.8**	66.2	78.8
2048	4-way	**60.0**	64.1	75.0
2048	8-way	**58.9***	62.8	73.4
4096	1-way	56.4	60.3	72.2
4096	2-way	**54.4**	57.2	65.0
4096	4-way	**53.5**	56.3	63.8
4096	8-way	**52.9***	**55.3**	61.8
8192	1-way	**50.5**	52.9	61.2
8192	2-way	**49.8**	51.5	56.7
8192	4-way	**49.0**	**50.2**	54.6
8192	8-way	**48.5***	**49.7**	53.9
16384	1-way	**45.3***	**46.9**	52.4
16384	2-way	**46.2**	**47.4**	50.6
16384	4-way	**46.3**	**46.9**	49.6
16384	8-way	**46.0**	**46.3**	48.4
32768	1-way	**41.4***	**42.8**	46.5
32768	2-way	**43.0**	44.3	46.5
32768	4-way	**43.3**	44.7	46.3
32768	8-way	**43.0**	44.5	45.8

Effective Access Times (ns) for Data Caches in ECL 100K with a miss penalty of 360.0 + 7.5*block_size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
2048	1-way	102.9	115.4	149.3
2048	2-way	90.0	98.9	124.0
2048	4-way	**85.6**	93.9	115.6
2048	8-way	**83.5***	91.2	112.3
4096	1-way	82.8	90.6	114.4
4096	2-way	75.3	80.9	96.6
4096	4-way	**72.7**	78.2	93.1
4096	8-way	**71.3***	76.1	89.3
8192	1-way	71.0	75.7	92.5
8192	2-way	66.0	69.4	79.9
8192	4-way	**63.6**	66.1	74.9
8192	8-way	**62.6***	**65.0**	73.3
16384	1-way	60.5	63.8	74.8
16384	2-way	**58.9**	61.4	67.7
16384	4-way	**58.2**	**59.5**	64.8
16384	8-way	**57.6***	**58.2**	62.5
32768	1-way	**52.9**	55.5	62.9
32768	2-way	**52.6**	55.0	59.5
32768	4-way	**52.1**	54.9	58.1
32768	8-way	**51.6***	54.6	57.1

Table 3-25. Times for ECL Data Caches.

Minimum ECL Cache Size where Direct-Mapped Caches have Similar or Smaller Effective Access Times			
Cache Type	Miss Time	5-Percent Size	Crossover Size
Mixed	fast	8K	32K
Instruction	fast	≤1K	≤1K
Data	fast	8K	16K
Mixed	slow	64K*	256K*
Instruction	slow	≤1K	32K
Data	slow	32K	256K*

Table 3-26. Crossover for ECL Caches.

This table shows the minimum cache size where the effective access time of a direct-mapped ECL cache is within five percent or is smaller than the smallest effective access time for a given cache size. While these sizes are slightly smaller for ECL than for TTL, the qualitative conclusions they yield are the same. Values marked with asterisks are cache sizes beyond my estimate of current, reasonable ECL cache sizes.

3.3.3.3. CMOS Caches

Finally I consider custom CMOS caches implemented on the same chip as a microprocessor. I report results for small caches (0.5K to 8K bytes) with shorter miss penalties than for TTL and ECL. I assume a fast miss penalty of one 50.0-ns-cycle plus one cycle per double-word, and a slow penalty of two cycles plus one cycle per word. These penalties are reasonable for on-chip caches interfaced to a board-level cache rather than to main memory. Cache access times are displayed in Table 3-27. CMOS effective access times are displayed in Figures 3-30 and 3-31 and Tables 3-28 through 3-31.

Degree of Associativity	Cache Access Time	
	(ns)	% increase from A=1
1	50.0	0.0
2	51.0	2.0
4	51.1	2.2
8	51.3	2.6

Table 3-27. CMOS Cache Access Times.

This table lists delays for set-associative caches similar to the SPUR instruction buffer, a 512-byte direct-mapped instruction cache implemented in 1.8 micron CMOS. Times are calculated by Duncombe [Dunc86] and discussed in Section 3.5. Since these VLSI assumptions are radically different than the MSI ones, comparing CMOS results with TTL or ECL results is difficult.

CMOS results are different from TTL and ECL results for mixed and data caches, because they show that direct-mapped caches yield significantly higher effective access times than do set-associative caches. For mixed caches with 16-byte blocks, for example, the effective access times of direct-mapped caches are 7 to 10 percent worse than set-associative times, assuming a *slow* miss penalty of 6 cycles. Direct-mapped caches perform worse, because of (1) the larger miss ratio difference between these smaller caches, and (2) the very small access time increases caused by adding set-associativity logic. The increases due to CMOS set-associativity logic are between 2.0 and 2.6 percent, whereas the increases for TTL and ECL range from 9.0 and 15 percent. Since my VLSI assumptions are radically different than my MSI ones, comparing CMOS results with TTL or ECL results is subject to more error. Nevertheless, while exact set-associative logic delays are subject to debate, it is clear that they should be smaller than MSI delays, since adding a multiplexor on a custom chip adds just the multiplexor delay, whereas adding a discrete multiplexor chip adds two chip crossings and possibly the delay required to encode signals for the multiplexor's *select* input.

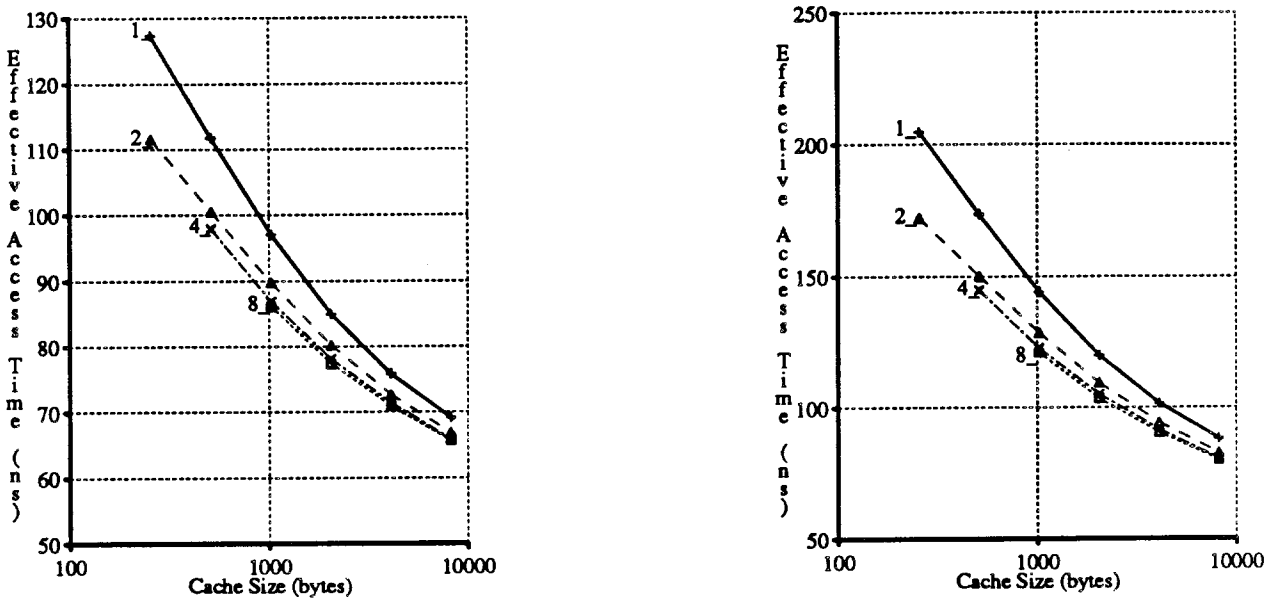


Figure 3-30. Effective Access Times for CMOS Mixed Caches.

This figure shows effective access times for custom CMOS mixed caches with 32-byte blocks. Cache access times of 50.0, 51.0, 51.1 and 51.3 ns for direct-mapped, two-, four- and eight-way set-associative caches are based on custom CMOS implementation assumptions from Section 3.5; off-chip miss penalties are 5 and 10 cycles. This data is displayed in tabular form in Table 3-29.

Direct-mapped effective access times are worse than set-associative ones regardless of miss penalty.

Results for CMOS instruction caches, on the other hand, are similar to results in the other technologies, namely, that direct-mapped instruction cache performance is similar to that of set-associative instruction caches. The behavior of instruction cache effective access times diverges from that of mixed and data caches, because the miss ratio spreads for small instruction caches diverge from those of small mixed and data caches.

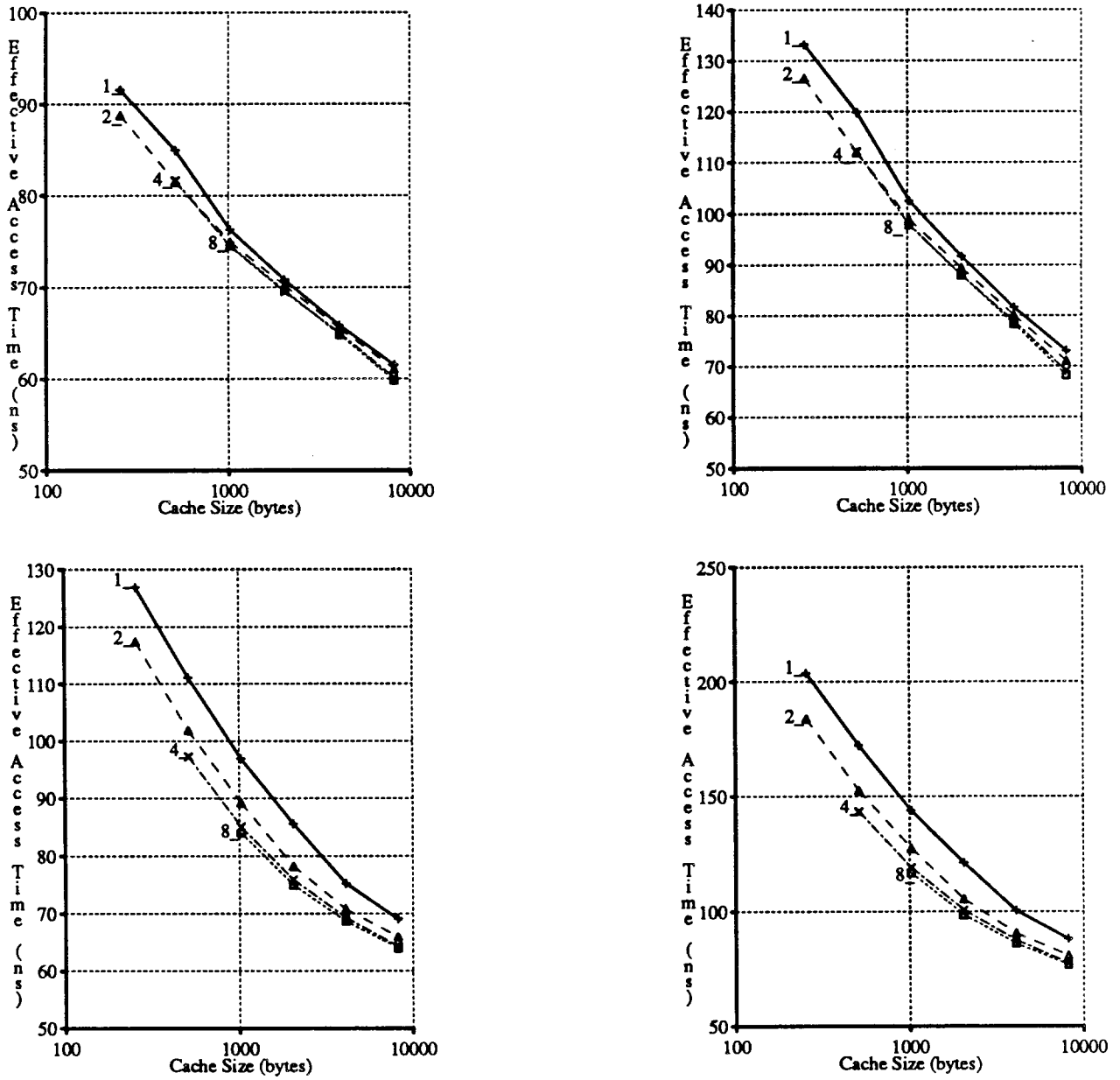


Figure 3-31. More CMOS Effective Access Times.

5-cycle misses (left), 10-cycle misses (right), I-caches (top) and D-caches (bottom).

Minimum CMOS Cache Size where Direct-Mapped Caches have Similar or Smaller Effective Access Times			
Cache Type	Miss Time	5-Percent Size	Crossover Size
Mixed	fast	4K	256K*
Instruction	fast	≤0.5K	64K*
Data	fast	8K	128K*
Mixed	slow	32K*	**
Instruction	slow	1K	**
Data	slow	16K*	**

Table 3-28. Crossover for CMOS Caches.

This table shows the minimum cache size where the effective access time of a direct-mapped CMOS instruction cache is within five percent or is smaller than the smallest effective access time for a given cache size. Values marked with asterisks are cache sizes beyond my estimate of current, reasonable custom CMOS cache sizes; while two asterisks indicate that no crossover occurs for caches sizes of 256K bytes and less.

For mixed and data caches, set-associative effective access times are smaller than direct-mapped ones for reasonable cache sizes. For instruction caches, all effective access times are similar.

Effective Access Times (ns) for Mixed Caches in Custom CMOS with a miss penalty of 50.0 + 6.25*block_size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
512	1-way	91.4	111.7	163.5
512	2-way	**86.8**	100.5	134.3
512	4-way	**86.0***	97.9	126.3
1024	1-way	82.7	97.0	132.0
1024	2-way	**79.5**	89.7	113.2
1024	4-way	**77.9***	86.9	108.9
2048	1-way	75.7	85.0	107.4
2048	2-way	**73.4**	80.2	96.7
2048	4-way	**72.0***	78.1	92.5
4096	1-way	**69.5**	75.8	91.4
4096	2-way	**67.8**	72.7	83.5
4096	4-way	**67.1***	71.5	81.1
8192	1-way	**64.7**	69.2	80.0
8192	2-way	**63.4**	66.9	74.5
8192	4-way	**62.6***	65.9	72.7

Effective Access Times (ns) for Mixed Caches in Custom CMOS with a miss penalty of 100.0 + 12.5*block_size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
512	1-way	132.7	173.5	276.9
512	2-way	**122.6**	150.1	217.7
512	4-way	**120.8***	144.7	201.6
1024	1-way	115.4	144.1	214.0
1024	2-way	**108.0**	128.5	175.3
1024	4-way	**104.7***	122.7	166.6
2048	1-way	101.4	119.9	164.8
2048	2-way	**95.7**	109.4	142.4
2048	4-way	**92.9***	105.1	133.8
4096	1-way	88.9	101.7	132.8
4096	2-way	**84.7**	94.3	116.1
4096	4-way	**83.0***	91.8	111.1
8192	1-way	79.5	88.3	109.9
8192	2-way	**75.8**	82.8	97.9
8192	4-way	**74.2***	80.7	94.4

Table 3-29. Times for CMOS Mixed Caches.

This table shows effective access times for CMOS mixed caches with 16-, 32- and 64-byte blocks, assuming a "fast" miss penalty (top) and a "slow" miss penalty (bottom). Miss ratios are based on warm-start simulations with trace average *2nd500k*, while cache access times on are based on CMOS implementation assumptions from Section 3.5. The smallest effective access time at each cache size is indicated with an asterisk; effective access times within five percent of this time are display in **bold.**

Effective Access Times (ns) for Instruction Caches in Custom CMOS with a miss penalty of 50.0 + 6.25**block size* ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
512	1-way	**79.5**	84.9	98.6
512	2-way	**77.8**	81.5	92.3
512	4-way	**77.2***	81.6	93.4
1024	1-way	**73.2**	76.3	84.5
1024	2-way	**72.5**	74.9	82.1
1024	4-way	**72.1***	74.5	81.3
2048	1-way	**68.4**	70.8	77.3
2048	2-way	**68.3**	70.2	75.4
2048	4-way	**67.7***	69.5	74.7
4096	1-way	**64.1**	65.8	70.8
4096	2-way	**63.9**	65.5	69.5
4096	4-way	**63.4***	65.0	68.9
8192	1-way	**60.2**	61.5	64.9
8192	2-way	**59.7**	61.0	64.1
8192	4-way	**58.7***	60.1	63.2

Effective Access Times (ns) for Instruction Caches in Custom CMOS with a miss penalty of 100.0 + 12.5**block size* ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
512	1-way	109.0	119.9	147.2
512	2-way	**104.7**	112.0	133.6
512	4-way	**103.3***	112.2	135.7
1024	1-way	**96.4**	102.5	118.9
1024	2-way	**93.9**	98.9	113.1
1024	4-way	**93.1***	97.9	111.5
2048	1-way	**86.7**	91.7	104.6
2048	2-way	**85.7**	89.4	99.8
2048	4-way	**84.3***	**88.0**	98.3
4096	1-way	**78.1**	81.7	91.5
4096	2-way	**76.7**	80.1	88.0
4096	4-way	**75.6***	**78.9**	86.8
8192	1-way	70.4	73.1	79.9
8192	2-way	**68.3**	71.1	77.2
8192	4-way	**66.4***	**69.1**	75.2

Table 3-30. Times for CMOS Instruction Caches.

Effective Access Times (ns) for Data Caches in Custom CMOS with a miss penalty of 50.0 + 6.25*block_size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
512	1-way	87.1	111.1	172.7
512	2-way	**82.4**	101.8	146.8
512	4-way	**79.9***	97.2	138.5
1024	1-way	79.3	96.9	137.6
1024	2-way	**75.0**	89.2	120.4
1024	4-way	**72.5***	85.0	114.1
2048	1-way	72.8	85.6	113.9
2048	2-way	**68.7**	78.2	99.5
2048	4-way	**67.1***	75.9	94.6
4096	1-way	66.5	75.2	95.2
4096	2-way	**64.1**	70.8	84.8
4096	4-way	**63.1***	69.3	82.5
8192	1-way	**62.8**	69.1	83.5
8192	2-way	**61.2**	66.0	75.9
8192	4-way	**60.2***	64.3	72.8

Effective Access Times (ns) for Data Caches in Custom CMOS with a miss penalty of 100.0 + 12.5*block_size ns				
Cache Size (bytes)	Degree of Associativity	Block Size (bytes) 16	32	64
512	1-way	124.3	172.2	295.4
512	2-way	**113.8**	152.6	242.6
512	4-way	**108.8***	143.4	225.9
1024	1-way	108.6	143.8	225.2
1024	2-way	98.9	127.4	189.9
1024	4-way	**93.9***	119.0	177.2
2048	1-way	95.6	121.1	177.8
2048	2-way	**86.3**	105.5	148.0
2048	4-way	**83.1***	100.6	138.1
4096	1-way	83.0	100.5	140.4
4096	2-way	**77.1**	90.5	118.6
4096	4-way	**75.0***	87.6	114.0
8192	1-way	75.6	88.1	116.9
8192	2-way	**71.3**	80.9	100.7
8192	4-way	**69.3***	77.5	94.5

Table 3-31. Times for CMOS Data Caches.

3.3.4. A Hybrid Design

The evidence presented so far suggests that direct-mapped caches have faster access times while set-associative caches have lower miss ratios. A cache design that tries to exploit both these advantages is called an *MRU* (most-recently-used) cache (see discussion of [Chan87] in Section 3.1.3). An MRU cache provides fast access to the most-recently-used block in a every set, and relatively slower access to other blocks. I refer to the collection of blocks that can be rapidly accessed as the *MRU region*. The MRU region for an n-way set-associative cache of size c has the same miss ratio as a direct-mapped cache of size c/n. For this reason, the effective access time of an MRU cache can be modeled as:

$$t_{eff}(C_{MRU}(C=c, A=n)) =$$

$$t_{MRU} + m(C(C=c/n, A=1)) * t_{LRU} + m(C(C=c, A=n)) * t_{memory}$$

where t_{MRU} is the access time of the MRU region, t_{LRU} is the additional time required to reach the non-MRU blocks, t_{memory} is the miss penalty, and $m(C(C=i, A=j))$ is the miss ratio of a j-way set-associative cache of size i.

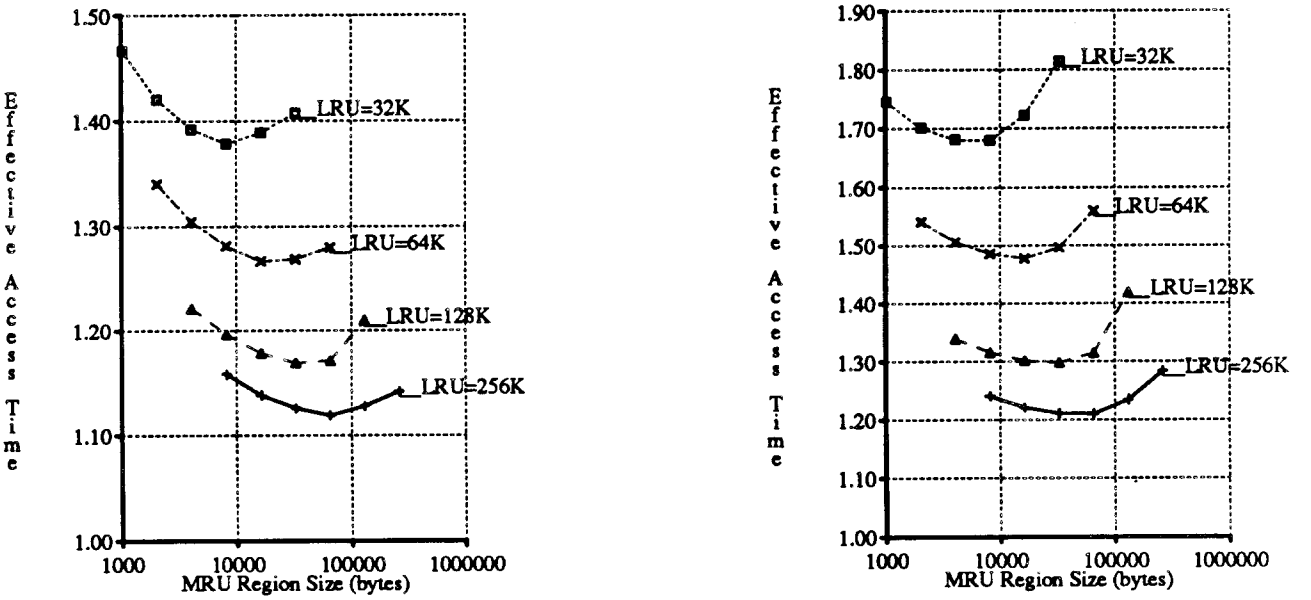


Figure 3-32. Times for MRU Mixed Caches.

This figure shows effective access times for MRU mixed caches with 32-byte blocks, based on warm-start miss ratios from trace average *2nd500k*. I assume that the MRU region is accessed in one cycle, one additional cycle is required to access the rest of the cache, and cache miss penalty is 10 (left) or 20 (right) additional cycles. The x-axis shows different MRU region sizes, while each line represents a different "LRU" cache size in bytes. The rightmost point in each line, where MRU and LRU sizes are equal, represents a regular direct-mapped cache. For example, the points on the line labeled "LRU=128K" from *right to left* represent a regular 128K-byte direct-mapped cache, a two-way set-associative 128K-byte cache with a 64K-byte MRU region, a four-way set-associative cache with a 32K-byte MRU region, etc.

A four-way set-associative MRU cache is best, but the advantage is small with the faster miss penalty.

Like direct-mapped and set-associative caches in the last section, I evaluate MRU caches with warm-start miss ratios from trace average *2nd500k*. Results for mixed caches, shown in Figure 3-32 corroborate the claim by Chang, Chao and So that the best organization for an MRU cache is four-way set-associative [Chan87]. It is not surprising that the minimum effective access time for MRU caches be found at such a small degree of associativity, since further increases in associativity reduce MRU size (and therefore increase the MRU miss ratio) without decreasing the LRU miss ratio much. Results for instruction caches and data caches are similar (see Figure 3-33).

While MRU caches have smaller effective access times than direct-mapped caches, the difference may not be important and can be overwhelmed by implementation disadvantages. Four-way set-associative MRU mixed caches, for example, have 2 to 3 percent better effective access times than direct-mapped caches with the fast miss penalty of 10 cycles, and 5 to 9 percent better times with the 20-cycle miss penalty. Thus, while MRU caches perform better than direct-mapped caches, they are worthwhile only if their implementation costs (chip area, board area, etc.) are not significantly larger than those of direct-mapped caches.

Furthermore in some implementations, the access time to a direct-mapped cache will be less than that to an MRU region, since the blocks of a direct-mapped cache are selected directly with address bits, whereas the blocks of an MRU region are selected indirectly. In an MRU cache, several address bits are

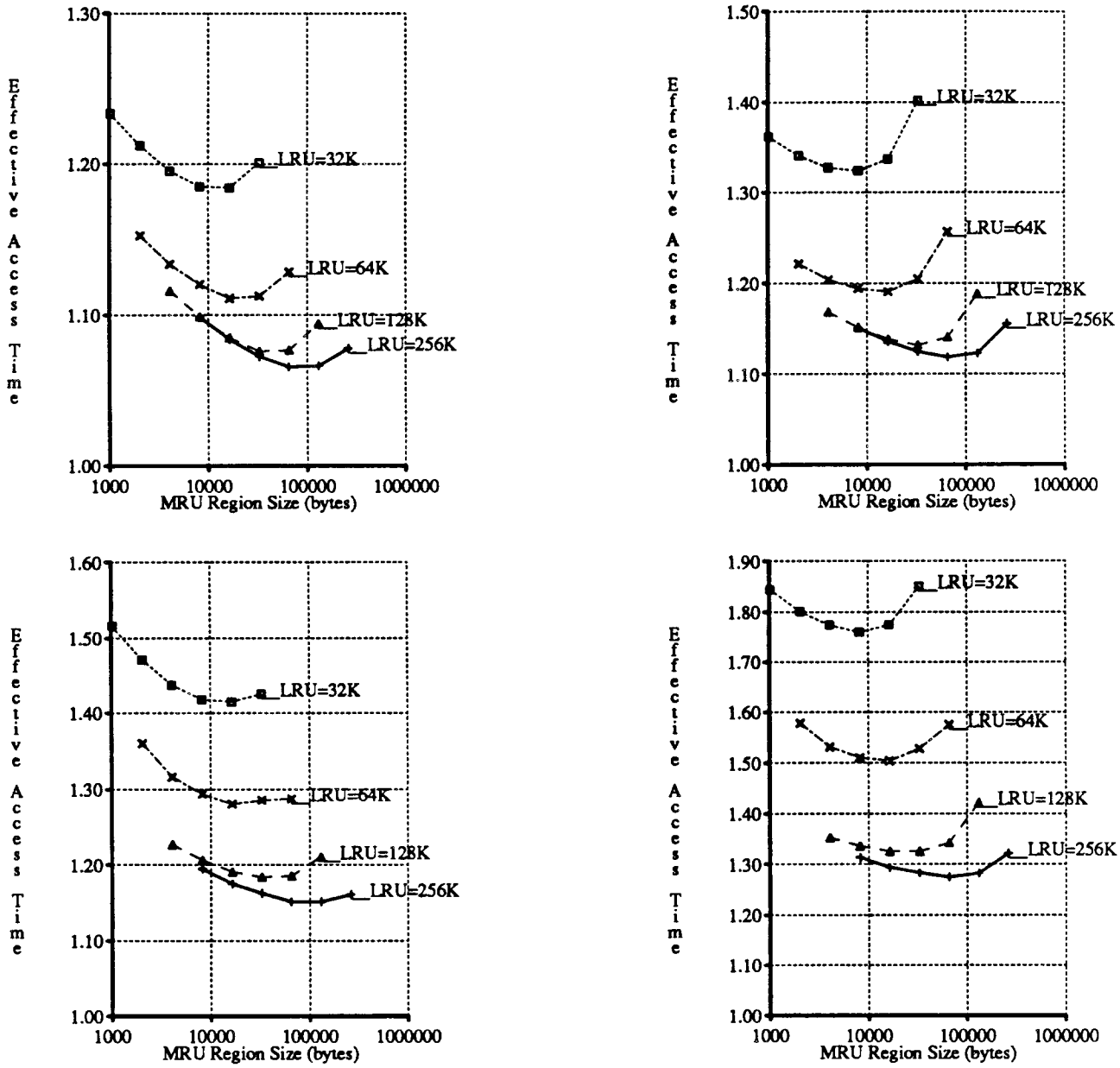


Figure 3-33. More MRU Effective Access Times.

Fast misses (left), slow misses (right), I-caches (top) and D-caches (bottom).

used to select $\log_2(a)$ bits associated with each set of a blocks; these bits are then used select the MRU block. Figure 3-34 shows that the advantage of MRU mixed caches is negated if they have 10 percent slower access times than direct-mapped caches. I expect access time degradations of 10 percent and larger in all MRU implementations where it is not possible to read out an entire set in parallel with reading the $\log_2(a)$ bits, Results for other degradations can be calculated from the miss ratios in Table 3-4.

An MRU cache may still be preferred to direct-mapped one if the cache must be physically tagged or if reducing memory traffic is paramount, as are the case for the system in [Chan87]. MRU caches are faster than large, physically-tagged direct-mapped caches, because an MRU cache allows parallel address translation, whereas the direct-mapped cache does not. An MRU cache produces less memory traffic than a direct-mapped cache, because the MRU cache's overall miss ratio is lower. In future

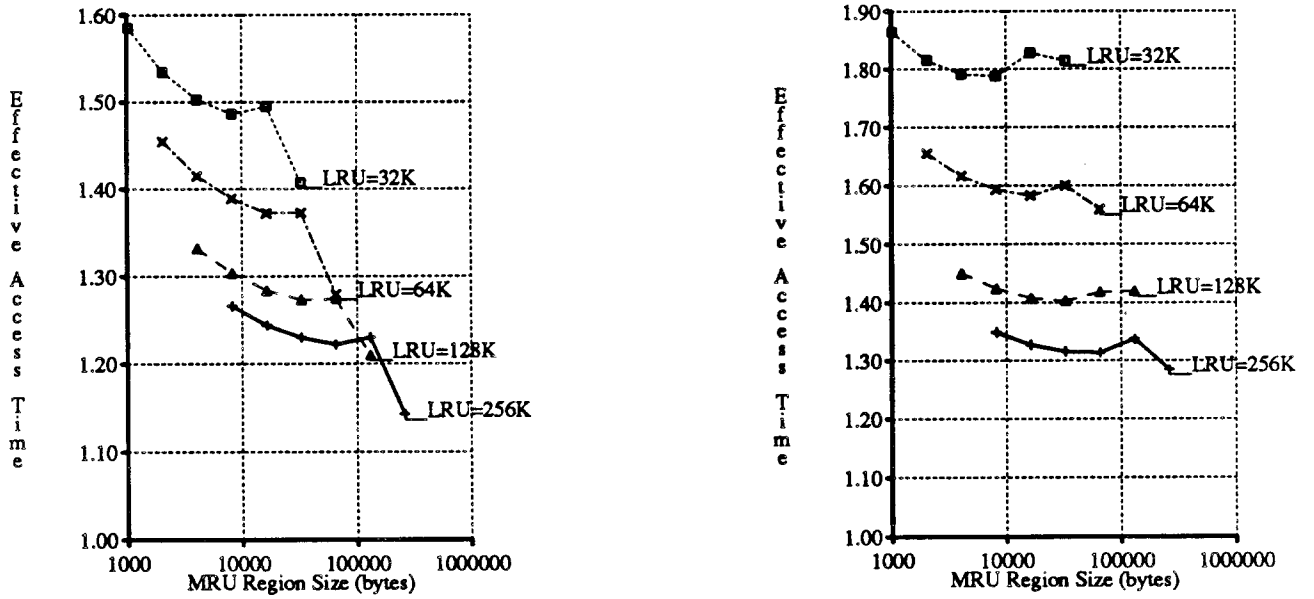


Figure 3-34. Effective Access Times for Slower MRU Caches.

This figure shows effective access times for MRU mixed caches with 32-byte blocks. Assumptions are similar to those of Figure 3-32, except that I set cycle times with MRU caches to be 10 percent slower than cycle times with direct-mapped caches to illustrate the advantage of MRU caches is negated if they are slower.

multiprocessor systems, however, it is not clear to me whether MRU cache will be preferred to two-level cache designs, where small, fast caches minimize access time, and large, slow caches minimize memory traffic.

3.4. Summary and Conclusions

I have studied cache associativity with miss ratios and effective access times. My miss ratio results confirm that increasing associativity reduces cache miss ratios by diminishing amounts. I find that the *ratios of miss ratios* between of caches of different associativities, *miss ratio spreads*, do not change rapidly as cache sizes increase. Consequently as caches get larger, the absolute miss ratio difference between caches of different associativities becomes smaller.

My effective access time results examine direct-mapped caches that have larger miss ratios, but shorter access times than set-associative caches. I show analytically that a direct-mapped cache yields smaller effective access time than a set-associative cache when the difference between their miss ratios times the miss penalty is less than the difference between their access times. I then use miss ratios from trace-driven simulation and implementation assumptions for TTL, ECL and custom CMOS caches to illustrate that direct-mapped caches can have smaller effective access times than set-associative caches in practice, as well as in theory. As cache size increases, I find that direct-mapped cache performance improves relative to set-associative cache performance, because the miss ratio difference gets smaller, but the access time difference does not. In particular, I find that direct-mapped caches have similar or better effective access times than set-associative caches at many cache configurations of 32K-bytes and larger. In many cases, however, effective access times are sufficiently close that the cache organization used should be based more on cost differences than on performance differences.

My effective access time work also underscores that implementation considerations should not be ignored, unless we can show their effects to be negligible. While it seems reasonable to assume that all caches of the same size have the same access time, doing so leads to misleading results as miss ratio differences decrease.

More work of this type is merited, since I expect memory hierarchies to become more, not less, important to future computers. In particular, extending effective access time results to hierarchies of caches and caches in multiprocessors is valuable. Cache hierarchies will become necessary as the ratio between processor speeds and main memory speeds increases. Caches are also necessary in shared-memory multiprocessors to reduce traffic in interconnection networks. In the future, caches at different points in the system may be optimized for different functions. I expect that analysis using trace data of existing systems and considering constraints of real implementation technologies will prove as important to these future problems as it has to the single-level caches I have addressed.

3.5. Appendix: Cache Implementations

This appendix examines implementing direct-mapped and set-associative caches in Advanced Schottky TTL, ECL and custom CMOS. I begin by examining how to implement set-associativity logic, which combines the results of N direct-mapped banks into the results of an N-way set-associative cache, in generic OR-gates, NOR-gates and multiplexors (MUXs). Then for each technology, I discuss a direct-mapped cache to use as an operating point, design the set-associativity logic, and determine the cache access times for direct-mapped and set-associative caches implemented in that technology.

Figure 3-35 shows possible implementations of the set-associativity logic for two-, four- and eight-way set-associative caches with OR-gates, NOR-gates and multiplexors. I use these descriptions in the rest of this appendix when I implement caches in specific technologies. The only non-trivial part of these designs is the implementation of *Select*, which appears to require an encoder. A simpler design is possible, however, since at most one bank can match.

Select can be implemented in a two-way set-associative cache by setting it equal to either *Match[1]* or *Match*[0]*, as the following truth table shows:

Match [0]	Match [1]	Select
0	0	x (cache miss)
1	0	0
0	1	1
1	1	x (illegal)

The truth table for *Select* in a four-way set-associative cache is:

Match [0]	Match [1]	Match [2]	Match [3]	Select	Select <0>	Select <1>
0	0	0	0	x	x	x
1	0	0	0	0	0	0
0	1	0	0	1	0	1
0	0	1	0	2	1	0
0	0	0	1	3	1	1

Select can be computed with two parallel two-input ORs:

$$Select\langle 0\rangle = Match[2] \text{ OR } Match[3]$$

$$Select\langle 1\rangle = Match[1] \text{ OR } Match[3],$$

or with two parallel two-input NORs:

$$Select\langle 0\rangle = Match[0] \text{ NOR } Match[1]$$

$$Select\langle 1\rangle = Match[0] \text{ NOR } Match[2].$$

The truth table for *Select* in an eight-way set-associative cache is:

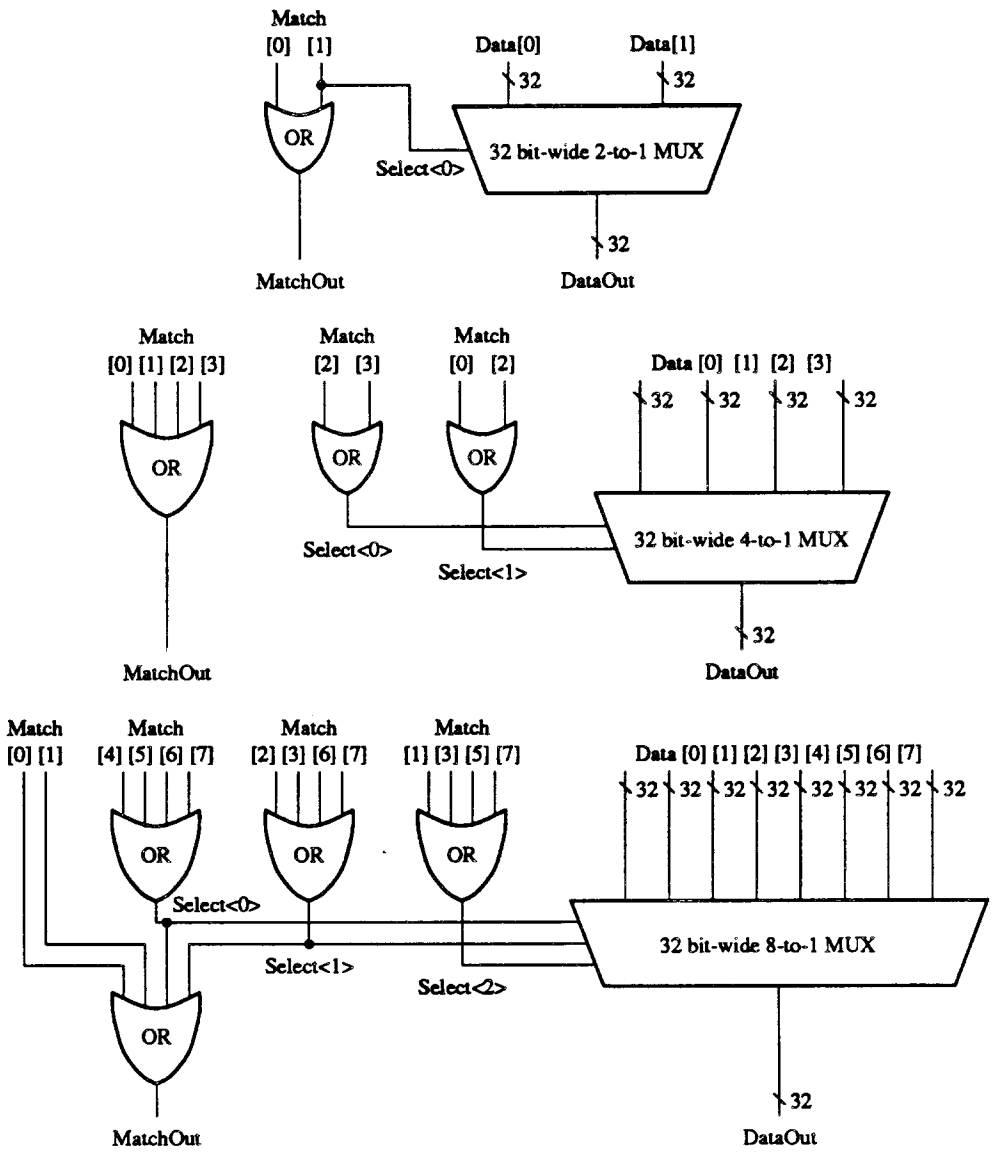


Figure 3-35. Set-Associativity Logic.

The logic needed to combine two (top), four (middle) and eight (bottom) direct-mapped banks into a set-associative cache.

Match								Select	Select		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]		<0>	<1>	<2>
0	0	0	0	0	0	0	0	x	x	x	x
1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	1
0	0	1	0	0	0	0	0	2	0	1	0
0	0	0	1	0	0	0	0	3	0	1	1
0	0	0	0	1	0	0	0	4	1	0	0
0	0	0	0	0	1	0	0	5	1	0	1
0	0	0	0	0	0	1	0	6	1	1	0
0	0	0	0	0	0	0	1	7	1	1	1

Select may be implemented in a single logic level of of 4-input ORs or NORs. With ORs:

Select<0> = *Match*[4] OR *Match*[5] OR *Match*[6] OR *Match*[7]

Select <1> = *Match* [2] OR *Match* [3] OR *Match* [6] OR *Match* [7]

Select <2> = *Match* [1] OR *Match* [3] OR *Match* [5] OR *Match* [7].

With NORs:

Select <0> = *Match* [0] NOR *Match* [1] NOR *Match* [2] NOR *Match* [3]

Select <1> = *Match* [0] NOR *Match* [1] NOR *Match* [4] NOR *Match* [5]

Select <2> = *Match* [0] NOR *Match* [2] NOR *Match* [4] NOR *Match* [6].

3.5.1. AS TTL Logic and Static CMOS RAMs

One technology for implementing a cache is off-the-shelf Advanced Schottky parts and static CMOS RAMs. This technology is attractive for implementing large board-level caches in personal workstations, as it is faster than other TTL families, but less power-consuming then ECL. I study caches in this technology by using the direct-mapped SPUR cache as an reference point, designing set-associativity logic, and combining the two together to form set-associative caches.

The SPUR cache contains 128K bytes of data and instructions, is organized into 4K direct-mapped 32-byte blocks tagged with virtual addresses, is implemented with AS TTL parts and CMOS static RAMs, and has an access time of 100 ns [Hill86, Wood86b]. The SPUR cache is somewhat more complex than the generic direct-mapped cache described in Section 3.3.2, because it converts process-specific virtual addresses to global virtual addresses, permits 40- and 64-bit accesses to aligned 64-bit words, and contains additional state information and logic for maintaining cache coherency. These differences, however, are not relevant to this analysis since they do not affect cache access time.

A two-, four- or eight-way set-associative cache can be constructed using the set-associativity logic of Figure 3-35, implemented with parts shown in Table 3-32, to connect two, four or eight directed-mapped banks.

The fastest way to compute *MatchOut* in a two-way set-associative cache is to use eight 74AS240 Octal Tri-State Buffers chips in the style of Figure 3-25. This implementation requires 9.0 ns for *Match[i]→DataOut* and 6.5 ns for *Data[i]→DataOut*. Alternatively eight 74AS158 Quad 2-to-1 MUXs can be used to achieve delays of 10.5 ns and 5.0 ns for *Match[i]→DataOut* and *Data[i]→DataOut*, respectively. Consequently, the tri-state buffers are preferred unless *Match[i]* is available more than 3.5 ns before *Data[i]*, which is unlikely.

The fastest way to compute *MatchOut* from *Match[0]* and *Match[1]* is with a 74AS02 2-input NOR gate in 4.5 ns†. Alternatively, the *Match[i]'s* can be wire-ORed if they are calculated with open-collector gates, such as the 74ALS33A 2-Input NOR. This solution is too slow, however, since even if the wire-ORing required no time, the open-collector gates produce *Match[i]* 7.5 ns later than the regular NORs (12.0 ns minus 4.5 ns).

The fastest way to compute *MatchOut* in a four-way set-associative cache is also with 74AS240 Octal Tri-State Buffer chips. While sixteen chips are required, the delays for *Match[i]→DataOut* and *Data[i]→DataOut* remain at 9.0 and 6.5 ns. Alternatively, one 74AS02 Quad 2-Input NOR and eight 74AS153 Quad 4-to-1 MUXs can be used, producing larger delays of 17.0 ns (4.5 ns plus 12.5 ns) and 8.0 ns with fewer chips (9 vs. 16). The smallest time for *Match[i]→MatchOut* is 6.0 ns, using a 74AS802 4-Input OR.

Similarly for an eight-way set-associative caches, using 32 74AS240 Octal Tri-State Buffer chips is faster than using four 74AS802 Triple 4-Input OR/NOR chips and 32 74AS151 Single 8-to-1 MUXs. With the multiplexors, the delays for *Match[i]→DataOut* and *Data[i]→DataOut* are 18.5 and 6.5 ns. The *Match[i]→DataOut* assumes that each *Select* bit is computed by four different gates, each driving eight loads, rather than being computed by one gate that must drive 32 loads. I also compute half of the

† In all these implementations I ignore the polarity of inputs and outputs.

Part Number	Part Name	Timing Path	Worst-Case Delay (ns)	Reference
74AS832A	Hex 2-Input OR	Data→DataOut	5.5	[Texa84], p. 2-545
74AS32	Quad 2-Input OR	Data→DataOut	5.8	[Texa84], p. 2-61
74AS02	Quad 2-Input NOR	Data→DataOut*	4.5	[Texa84], p. 2-11
74ALS33A	Quad 2-Input NOR w/ open-collector outputs	Data→DataOut*	12.0	[Texa84], p. 2-64 (high to low)
74AS802	Triple 4-Input OR /NOR	Data→DataOut Data→DataOut*	6.0 6.5	[Texa84], p. 2-51 (delay estimated)
74AS240	Octal Buffers w/ 3-state outputs	Data→DataOut* Select→DataOut*	6.5 9.0	[Texa84], p. 2-221
74AS158	Quad 2-to-1 MUX	Data→DataOut* Select→DataOut*	5.0 10.5	[Texa84], p. 2-135
74AS157	Quad 2-to-1 MUX	Data→DataOut Select→DataOut	6.0 11.0	[Texa84], p. 2-135
74AS153	Quad 4-to-1 MUX	Data→DataOut Select→DataOut	8.0 12.5	[Texa84], p. 2-128
74AS151	Single 8-to-1 MUX	Data→DataOut Data→DataOut* Select→DataOut Select→DataOut*	11.0 6.5 15.0 12.0	[Texa84], p. 2-124 [Nati84], p. 3-62

Table 3-32. Selected AS TTL Parts.

This table lists information on the Advanced Schottky OR/NOR gates useful for computing *MatchOut*, *Select*, and *DataOut* from the Texas Instruments TTL Data Book [Texa84] and the National Logic Data Book [Nati84]. *DataOut** on a multiplexor implies that the multiplexor inverts its output.

The TTL Data Book gives only typical delays for the 74AS802 Triple 4-Input OR/NOR, because this part has never been manufactured [Lutz86]. I choose to use the part anyway because doing associativity comparisons using a technology without a reasonable 4-input OR or NOR gate would unfairly bias results toward lower degrees of associativity. The worst-case delays used for this part are estimated by adding about 30 percent to the typical delays 4.5 ns and 5.0 ns for the OR and NOR [Lutz86].

The propagation delay for the 74ALS33 Quad 2-Input NOR with open collector output is 12.0 high to low and 33 ns low to high, assuming a 680 ohm pull-up and a 50 pF load. I report only the high-to-low time, because this part would only be used to compute *Select* active low.

Select signals with NOR rather than OR gates to reduce the maximum number of loads driven by any *Match[i]* from twelve to six loads. *MatchOut* can be computed in 12.0 ns with a single additional 74AS802 4-Input OR gate according to the following equation:

$$MatchOut = Match[0] \text{ OR } Match[1] \text{ OR } Select\langle 0\rangle \text{ OR } Select\langle 1\rangle.$$

Adding set-associative logic to the SPUR cache will increase cache access time by the longest delay path through the additional logic, since both *Data[0]* and *Match[0]* are available at the same time (in 100 ns), and both *DataOut* and *MatchOut* are required immediately. As Table 3-33 shows, the implementation of the set-associative logic with the shortest critical path is the one that uses tri-state buffers. Table 3-34 shows that using this implementation produces set-associative caches with 109.0 ns access times.

3.5.2. Emitter Coupled Logic

Another logic family suitable for high-performance computers is Emitter Coupled Logic (ECL). ECL is considerably faster and more power-hungry than AS TTL. ECL machines usually reside in machine rooms, because their cooling requirements are too great to be handled with typical office cooling. ECL has been the technology of choice for mainframes for many years, and is now also commonly-used in super-minicomputers. In addition, ECL has also been used in at least two personal workstations: the Xerox PARC Dorado [Pier83] and the DEC WRL Titan I [Bask86]. I study ECL

Degree of Associativity	Timing Path (ns)			Comments
	Match[i] →MatchOut	Match[i]→Select →DataOut	Data[i] →DataOut	
1	0.0	n/a	0.0	no logic
2	4.5	9.0	6.5	OR, 3-State
2	4.5	10.5	5.0	OR, MUX
4	6.0	9.0	6.5	OR, 3-State
4	6.0	17.0	8.0	OR, MUX
8	12.0	9.0	6.5	OR, 3-State
8	12.0	18.5	6.5	OR, MUX

Table 3-33. Cache Timing Paths with AS TTL.

This table presents delays for the three timing paths through Advanced Schottky TTL set-associativity logic. There is no path from *Match[i]* to *DataOut* in a direct-mapped cache. All *MatchOut*'s are computed with an explicit OR or NOR-gate ("OR"). *DataOut* can be computed with tri-state buffers ("3-State") or a multiplexor ("MUX").

Degree of Associativity	Cache Access Time	
	(ns)	% increase from A=1
1	100.0	0.0
2	109.0	9.0
4	109.0	9.0
8	109.0	9.0

Table 3-34. TTL Cache Access Times.

This table lists delays for set-associative caches that are similar to the SPUR cache. The SPUR Cache is 128K bytes, organized as 4K direct-mapped 32-byte blocks. It has a access time of 100 ns with a critical path to *Match*. Since *MatchOut* and *DataOut* are required at the same time, the additional logic for set-associativity increases cache access time by the time for the *Match[i]→Select→DataOut* timing path. The length of this path is minimized by using tri-state drivers rather than explicit multiplexors. Caches with associativities two, four and eight have the same access times, 109 ns, because, to first-order, the delay through tri-state buffers is not affected by how many of them are operating in parallel.

caches by using a modified version of the caches in the Titan I as a reference point and designing set-associativity logic.

The Titan I contains two identical 16K-byte ECL caches: one for instructions and one for data. Each cache is four-way set-associative with 16-byte blocks and uses random replacement [Niel86]. Address translation with 4k-byte pages is done in parallel with a cache lookup using a TLB per cache. The Titan I cache access time is 45 ns, using four rows of eight 15 ns 1K x 4 static ECL RAMs. Data from these chips can be combined with a four-to-one multiplexor, controlled either by address bits (direct-mapped) or by the *Match[i]*'s (four-way set-associative). The designers chose to implement the four-way set-associative cache, because it had a lower miss ratio, control of the multiplexor was not on the critical path in this implementation, and it restricted the cache index bits to be within the page offset, making it straightforward to do the cache lookup in parallel with address translation. Nielson reports the critical path is instead through the tag memory and match logic to the control that stalls the pipeline on a cache miss.

The reference point I use for ECL is a Titan-I-like direct-mapped cache with a 30 ns access time. I assume the faster access time principally because ECL RAMs are now faster. In 1986 for example, the average access time for the four fastest 1K x 4 static ECL RAMs from different manufacturers was 8.6 ns [Hear86]. Even faster RAMs with further exaggerate access time differences between direct-mapped and set-associative caches. To facilitate comparisons between TTL and ECL I assume, as I did

for TTL, that both paths to *MatchOut* and *DataOut* are critical.

The set-associativity logic for a two-way set-associative ECL cache can be implemented four ways: using 2-to-1 MUX/Latches, using OR-AND-INVERT gates, using selectively enabled Drivers, or using chip-select on the data memory chips. In all cases *MatchOut* can be computed in zero-time by wire-ORing together *Match[0]* and *Match[1]*, making the delay for *Match[i]→MatchOut* 0.00 ns. Table 3-35 shows selected ECL parts, useful for building set-associativity logic.

Part Number	Part Name	Timing Path	Worst-Case Delay (ns)	Reference
F100102	Quint 2-Input OR /NOR	Data→DataOut Enable→DataOut	1.40 2.20	[Fair86], p. 3-7
F100101	Triple 5-Input OR /NOR	Data→DataOut	1.30	[Fair86], p. 3-5
F100117	Triple 2-Wide OR-AND /OR-AND-INVERT	Data→DataOut Enable→DataOut	2.60 1.40	[Fair86], p. 3-22
F100113	Quad Driver	Data→DataOut Enable→DataOut	1.40 1.90	[Fair86], p. 3-16
F100155	Quad MUX/Latch	Data→DataOut Select→DataOut Enable→DataOut	1.40 3.50 2.50	[Fair86], p. 3-96 (transparent mode)
F100171	Triple 4-Input MUX	Data→DataOut Select→DataOut Enable→DataOut	1.70 3.00 2.40	[Fair86], p. 3-142
F100163	Dual 8-Input MUX	Data→DataOut Select→DataOut	1.80 3.10	[Fair86], p. 3-118

Table 3-35. Selected ECL 100K Parts.

This table lists information on the ECL 100K parts useful for computing *MatchOut*, *Select*, and *DataOut* from the F100K ECL Data Book [Fair85]. All parts generate dual-rail outputs with symmetric propagation delay. Since ECL gates drive outputs high and require terminating resistors to pull outputs low, signals can be ORed by wiring them together. Worst-case case delays assume a dual in-line package, V_{EE} = -4.2 V to -4.8 V, and a temperature of 85°C. Delays for the F100155 Quad MUX/Latch assume that the latch is enabled so that data just passes through it (transparent mode).

The first approach for computing *DataOut* is to use eight F100155 Quad MUX/Latches even though the latch is not used, i.e., it is left in transparent mode. *Select* is equal to either *Match[1]* or *Match*[0]*. If I use *Match[1]* to drive *Select* on four multiplexors, and *Match*[0]* to drive the other four, then no additional buffering is required, and *DataOut* can be computed with a delay of 3.50 ns from *Select* and a delay of 1.40 ns from *Data*. The delays through this logic are 3.50 ns for *Match[i]→Select→DataOut* and 1.40 ns for *Data[i]→DataOut*.

A second way to implement the additional logic for a two-way set-associative cache is to build a 32-bit wide 2-to-1 multiplexor from eleven F100117 Triple 2-Wide OR-AND-INVERT chips. The equation for a 2-to-1 multiplexor is:

$$DataOut = (Data[0] \text{ AND } Select^*) \text{ OR } (Data[1] \text{ AND } Select).$$

If *Data*[0]* and *Data*[1]* are available, the same function can be implemented with an OR-AND-INVERT gate as follows:

$$DataOut = [(Data^*[0] \text{ OR } Select) \text{ AND } (Data^*[1] \text{ OR } Select^*)]^*.$$

Since these gates have a propagation delay of 2.60 ns, it appears that they are faster than the MUX/Latch whose delay is 3.50 ns from *Select*. However, it is not possible to have *Match[1]* and *Match*[0]*, which are both equal to *Select*, drive 16 loads each. Consequently, one level of buffering with four F100113 Quad Drivers is necessary so that each *Match* signal drives three loads, and each *Select* or *Select** signal drives at most three loads. This buffering adds 1.40 ns to the delay, making this solution 0.10 ns slower than using the MUX/Latches, which requires about half as many chips.

A third way to implement the additional logic for a two-way set-associative cache is to build two 32-bit wide "tri-state" buffers (see Figure 3-25) from sixteen F100113 Quad Driver chips. This design uses eight drivers per bank. The 32-bits of *Data[i]* must be connected to the data inputs and *Match[i]* connected to the enables. The corresponding bits of the driver's output must then be connected together to produce *DataOut** by a wired-OR. The delays through this logic are 1.90 ns for *Match[i]→Select→DataOut* and 1.40 ns for *Data[i]→DataOut*. This method is faster than the first two, but uses more chips.

The final way to implement the 32-bit wide multiplexor is with chip-select on the data RAMs. In this design, the outputs of the two banks of data memory chips, *Data[0]* and *Data[1]*, are connected directly to *DataOut*. On a cache lookup, the data memory chips are addressed, but not enabled. At the same time, the tag memory is accessed and *Match[0]* and *Match[1]* computed. *Match[i]* is then distributed to the chip-selects of all data memory chips in bank *i*. The delay from *Match[i]* to *DataOut* for this technique is, therefore, the data memory chip delay from chip-select plus buffer delays incurred to distribute each *Match[i]*. This delay is greater than the 3.5 ns required by the explicit 2-to-1 multiplexor design, since the delay from chip select required by most static ECL RAMS is at least that large [Hear86]. Nevertheless, multiplexing using chip-select may be attractive, since it saves chips and board area, but one must be aware of the limited drive capability of many RAM chips.

The set-associativity logic for a four-way set-associative cache can be computed using multiplexors or drivers. The multiplexor implementation uses 12 ECL 100K parts: one F100102 Quint 2-Input OR/NOR, and eleven F100171 Triple 4-Input MUXs. *Select* can be computed with two 2-Input OR/NOR (or with 5-input OR/NOR) gates in 1.40 ns (or 1.30 ns). However, each select bit must drive 11 loads. Instead, I can compute *Select* twice in parallel with four 2-Input OR/NOR gates so that each gate drives no more than six loads. *DataOut* requires the eleven Triple 4-Input MUXs which have a delay of 3.00 ns from *Select* and a delay of 1.70 ns from *Data*. The delays through this logic are, therefore, 4.40 ns for *Match[i]→Select→DataOut* and 1.70 ns for *Data[i]→DataOut*.

As with the two-way set-associative cache, *DataOut* can be computed using F100113 Drivers. The delays through this logic are 1.90 ns for *Match[i]→Select→DataOut* and 1.40 ns for *Data[i]→DataOut*. However, 32 F100113 Driver chips are required (eight chips per bank times four banks).

The set-associativity logic for an eight-way set-associative cache can also be computed using multiplexors or drivers. The multiplexor implementation uses 18 ECL 100K parts: two F100101 Triple 5-input OR/NOR and 16 F100163 Dual 8-Input MUXs. *Select* can be computed with one level of three 5-Input OR/NOR gates in 1.30 ns. I can reduce the fan-out from the select bits from 16 to 8 by using three NORs on a second F100101 to drive half of the multiplexor selects. *DataOut* requires the 16 Dual 8-Input MUXs with a delay of 3.10 ns from *Select* and a delay of 1.80 ns from *Data*. The delays through this logic are 4.40 ns for *Match[i]→Select→DataOut* and 1.80 ns for *Data[i]→DataOut*.

Again, *DataOut* can be computed using F100113 Drivers. The delays through this logic are 1.90 ns for *Match[i]→Select→DataOut* and 1.40 ns for *Data[i]→DataOut*. However, 64 F100113 Driver chips are required (eight chips per bank times eight banks).

I now compute ECL cache access times by combining the assumptions for my direct-mapped operating point with the delays for the set-associativity logic, shown in Table 3-36. Table 3-37 gives these access times with the set-associativity logic implemented with explicit multiplexors, rather than with F100113 driver chips. I discard using the driver chips, since doing so adds 16, 32 and 64 chips to two-, four- and eight-way set-associative cache designs. This number of additional chips can be equal to the number of memory chips in the cache, since ECL RAMS are often four bits wide. Using multiplexors, on the other hand, is more reasonable, since doing so adds only 8, 12, and 17 chips, allowing a larger or cheaper cache to be implemented.

Degree of Associativity	Timing Path (ns)			Comments
	Match[i] →MatchOut	Match[i]→Select →DataOut	Data[i] →DataOut	
1	0.00	n/a	0.00	no logic
2	0.00	3.50	1.40	wired-OR, MUX
2	0.00	1.90	1.40	wired-OR, Drivers
4	0.00	4.40	1.70	wired-OR, MUX
4	0.00	1.90	1.40	wired-OR, Drivers
8	0.00	4.40	1.80	wired-OR, MUX
8	0.00	1.90	1.40	wired-OR, Drivers

Table 3-36. Cache Timing Paths with ECL 100K.

This table presents delays for the three timing paths through the additional set-associativity logic when the logic is implemented with ECL 100K. There is no path from *Match[i]* to *DataOut* in a direct-mapped cache. *MatchOut* is computed in zero-time by wiring the *Match[i]*'s together ("wired-OR"). *DataOut* can be computed with a multiplexor ("MUX") or with drivers whose outputs are wired together ("Drivers").

Degree of Associativity	Cache Access Time	
	(ns)	% increase from A=1
1	30.0	0.0
2	33.5	11.7
4	34.4	14.7
8	34.4	14.7

Table 3-37. ECL Cache Access Times.

This table lists delays for set-associative caches that are similar to the SPUR design except that they are implemented in ECL with a 30 ns access time. In particular, this means that the critical path through the additional set-associative logic is through the *Match[i]→Select→DataOut* timing path. I assume a 30 ns access time rather than the 45 ns access time of the Titan I, because faster static ECL RAMs are now available.

3.5.3. Custom CMOS

CMOS is commonly used to implement microprocessors, because it consumes low static power and can be fabricated in large chips with good density. Most future microprocessors will include some on-chip cache memory to reduce memory reference latencies and off-chip data and instruction traffic (see Chapter 4). A cache on a processor chip will, of course, have to be implemented in the same technology as the rest of the chip. Consequently, I expect custom CMOS caches to be common at the top of microprocessor memory hierarchies. As with TTL and ECL, I study CMOS caches by defining a direct-mapped operating point, designing set-associativity logic, and combining the two to form set-associative caches.

I use the instruction buffer (IB) on the SPUR CPU chip as my CMOS cache reference point. The SPUR IB, described in detail in Chapter 4, is an instruction cache that contains 512 bytes (128 instructions) organized into 16 direct-mapped blocks of eight instructions each. The SPUR IB implementation is described by Duncombe [Dunc86] and in the appendix of Chapter 4. It has an access time of 50 ns and a cycle time of 100 ns. An instruction cache similar to the SPUR IB, but designed in isolation, can have a shorter access time, because the SPUR IB had to conform to aspect ratio (the ratio of length to width of an implementation) and timing constraints dictated by the rest of the CPU. Nevertheless, I assume a 50 ns access time for my direct-mapped CMOS reference point, since no useful cache is designed in isolation. Agarwal et al. for example, also found that implementation concerns, such as aspect ratio, were important to the design of the MIPS-X CMOS instruction cache [Agar87a].

The design of set-associativity logic for custom CMOS differs from the logic shown in Figure 3-35 whenever signals do not have to cross chip boundaries. These differences affect access time in three places. First, a multiplexor in AS TTL and ECL 100K is a separate chip, so increasing associativity from direct-mapped adds to the critical path the time for the multiplexor circuit plus the time to go on and off the multiplexor chip. In custom CMOS, increasing the associativity only adds the delay for the multiplexor circuit. Consequently, increasing associativity has less of a negative effect on cache access time in custom CMOS than in AS TTL or ECL.

Second, the *Select* inputs for an N-to-1 multiplexor in custom CMOS need not be encoded, into $\log_2(N)$ signals, because running the N *Match[i]* 's to the multiplexor takes only one, two, and five more wires for N equal to two, four, and eight. However, standard multiplexor chips always encode these signals to save scarce pins even though the signals are often decoded once on the multiplexor chip. In this respect the output of an on-chip multiplexor is selected they same way a tri-state buffer is enabled.

Finally, if the cache and the CPU are on the same chip, the N *Match[i]* 's do not have to be explicitly ORed together to form *MatchOut*, rather all of the *Match[i]* 's can be passed in the execution unit control where the OR delay can often be hidden in other overheads. In contrast when the execution unit is on a different chip(s), the *Match[i]* 's are usually ORed together to save pins.

For the reasons given above, Duncombe found the delay for set-associative logic is small (see Table 3-38) [Dunc86]. I assume that a set-associative CMOS cache will be slower than a direct-mapped one due to these delays (see Table 3-39). On average I expect this to be true, but in any given implementation, other implementation changes that accompany adding set-associativity, e.g., altering the aspect ratio of the data array, can cause larger positive or negative changes in access time.

Degree of Associativity	Timing Path (ns)		
	Match[i] →MatchOut	Match[i]→Select →DataOut	Data[i] →DataOut
1	0.0	n/a	0.0
2	0.0	1.0	1.0
4	0.0	1.1	1.2
8	0.0	1.3	1.4

Table 3-38. Cache Timing Paths with Custom CMOS.

This table presents delays for the three timing paths through the additional set-associativity logic when the logic is implemented in custom CMOS. Results are based on multiplexor delays calculated by Duncombe [Dunc86]. I assume that ORing the *Match[i]* 's can be hidden in control. Thus, the delays for all *Match[i]→MatchOut* paths are zero. I further assume that all multiplexor select bits are not encoded. Thus, the delays for all *Match[i]→Select→DataOut* paths are just the multiplexor delays from select. Finally, as always, the delay for all *Data[i]→DataOut* paths is the multiplexor delay from *Data[i]*.

Degree of Associativity	Cache Access Time	
	(ns)	% increase from A=1
1	50.0	0.0
2	51.0	2.0
4	51.1	2.2
8	51.3	2.6

Table 3-39. CMOS Cache Access Times.

This table lists delays for set-associative caches similar to the SPUR instruction buffer, a 512-byte direct-mapped instruction cache implemented in 1.8 micron CMOS.

3.6. References

[Agar86] A. Agarwal, R. L. Sites and M. Horowitz, ATUM: A New Technique for Capturing Address Traces Using Microcode, *Proc. Thirteenth International Symposium on Computer Architecture* (June 1986).

[Agar87a] A. Agarwal, P. Chow, M. Horowitz, J. Acken, A. Salz and J. Hennessy, On-chip Instruction Caches for High Performance Processors, *Proc. Conf. on Advanced Research in VLSI*, Stanford (March 1987).

[Agar87b] A. Agarwal, M. Horowitz and J. Hennessy, Cache Performance of Operating Systems and Multiprogramming Workloads, submitted to Trans. Computer Systems (April 1987).

[Alex86] C. Alexander, W. Keshlear, F. Cooper and F. Briggs, Cache Memory Performance in a UNIX Environment, *Computer Architecture News*, 14, 3 (June 1986), 14-70.

[Bask86] F. Baskett, *Titan I*, Dec. WRL, (January 1986). U.C., Berkeley Computer System Seminar.

[Bell74] J. Bell, D. Casasent and C. G. Bell, An Investigation of Alternative Cache Organizations, *IEEE Trans. on Computers*, C-23, 4 (April 1974), 346-351.

[Cham83] J. M. Chambers, W. S. Cleveland, B. Kleiner and P. A. Tukey, *Graphical Methods for Data Analysis*, Duxbury Press, Boston, (1983).

[Chan87] J. H. Chang, H. Chao and K. So, Cache Design of a Sub-Micron CMOS System/370, *14th Annual International Symposium on Computer Architecture*, Pittsburgh, PA (June 1987), 208 - 213.

[Cho86] J. Cho, A. J. Smith and H. Sachs, The Memory Architecture and the Cache and Memory Management Unit for the Fairchild CLIPPER Processor, Computer Science Division Technical Report UCB/Computer Science Dpt. 86/289, University of California, Berkeley (April, 1986).

[Clar83] D. W. Clark, Cache Performance in the VAX-11/780, *ACM Trans. on Computer Systems*, 1, 1 (February, 1983), 24 - 37.

[Dion86] J. Dion, *Private Communication*, Dec. WRL, (May 1986).

[Dunc86] R. R. Duncombe, The SPUR Instruction Unit: An On-Chip Instruction Cache Memory for a High Performance VLSI Multiprocessor, Unpublished Master's Report, University of California, Berkeley (August, 1986).

[East75] M. C. Easton, Measuring Cold-Start Miss Ratios, Computer Science RC 5692 (#24518), IBM Watson Research Center (October, 1975).

[Fair85] Fairchild, F100K ECL Data Book (1985).

[Haik84] I. J. Haikala and P. H. Kutvonen, Split Cache Organizations, CS Report C-1984-40., Univ. of Helsinki (August 1984).

[Hear86] Hearst Business Communications, Inc., IC Master, Vol. II (1986).

[Hill86] M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, Design Decisions in SPUR, *IEEE Computer*, 19, 11 (November 1986).

[Joup86] N. Jouppi, *Private Communication*, Dec. Western Research Lab, (December 1986).

[Kapl73] K. R. Kaplan and R. O. Winder, Cache-based Computer Systems, *Computer*, 6, 3 (March, 1973).

[Lipt68] J. S. Liptay, Structural Aspects of the System/360 Model 85, Part II: The Cache, *IBM Systems Journal*, 7, 1 (1968), 15-21.

[Lutz86] K. Lutz, *Private Communication*, University of California, Berkeley, (July 1986).

[MacG85] D. MacGregor and J. Rubinstein, A Performance Analysis of MC68020-based Systems, *IEEE MICRO*, 5, 6 (December 1985).

[Matt70] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger, Evaluation techniques for storage hierarchies, *IBM Systems Journal*, 9, 2 (1970), 78 - 117.

[Matt71] R. L. Mattson, Evaluation of Multilevel Memories, *IEEE Trans. on Magnetics*, MAG-7, 4 (December 1971).

[Nati84] National Semiconductors, National Logic Data Book (1984).

[Niel86] M. Nielson, *Private Communication*, Dec. Western Research Lab, (August 1986).

[Pier83] K. A. Pier, A Retrospective on the Dorado, A High-Performance Personal Computer, *Proc. Tenth Symposium on Computer Architecture* (June 1983), 252-269.

[Siew82] D. P. Siewiorek, C. G. Bell and A. Newell, Computer Structures: Principles and Examples, McGraw Hill (1982).

[Smit78] A. J. Smith, A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory, *IEEE Trans. on Software Engineering*, SE-4, 2 (March 1978), 121-130.

[Smit82] A. J. Smith, Cache Memories, *Computing Surveys*, 14, 3 (September, 1982), 473 - 530.

[Smit83] J. E. Smith and J. R. Goodman, A Study of Instruction Cache Organizations and Replacement Policies, *Proc. Tenth International Symposium on Computer Architecture*, Stockholm, Sweden (June 1983), 132-137.

[Smit85] A. J. Smith, Cache Evaluation and the Impact of Workload Choice, *Proc. Twelfth International Symposium on Computer Architecture* (June 1985).

[Smit87] A. J. Smith, Line (Block) Size Choice for CPU Caches, *IEEE Trans. on Computers*, C-36, 9 (September 1987).

[Stre76] W. D. Strecker, Cache Memories for PDP-11 Family Computers, *Proc. Third International Symposium on Computer Architecture* (January 1976), 155-158.

[Stre83] W. D. Strecker, Transient Behavior of Cache Memories, *ACM Trans. on Computer Systems*, 1, 4 (November, 1983).

[Texa84] Texas Instruments, Inc., The TTL Data Book, Vol. 3, Advanced Low-Power Schottky, Advanced Schottky (1984).

[Wood86a] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, J. Pendleton, S. A. Ritchie, R. H. Katz and D. A. Patterson, An In-Cache Address Translation Mechanism, *Proc. Thirteenth International Symposium on Computer Architecture*, Tokyo, Japan (June 1986).

[Wood86b] D. A. Wood, S. J. Eggers and G. Gibson, SPUR Memory System Architecture, Unpublished Report, University of California, Berkeley (August, 1986).

4 Instruction Memory on a Single-Chip RISC

4.1. Introduction

High-speed instruction memories, which hold instructions expected to be executed soon, are important to all computers, because they can cost-effectively bolster performance by reducing delays owing to instruction fetches. Instruction memories on single-chip RISC microprocessors are particularly important, because RISC chips must fetch numerous, simple instructions over a limited inter-chip datapath. Here I examine and compare two instruction memory organizations: *instruction buffers* (IBs), which are conventional instruction caches, and *target instruction buffers* (TIBs), which are buffers that hold one or more instructions at recent branch targets. TIBs should not confused with *branch target buffers* (BTBs). TIBs map branch target addresses to instructions at branch targets to reduce instruction fetch latency, while BTBs map branch instruction addresses to branch target addresses to reduce pipeline bubbles.

I next discuss how and why instruction memories have been used, why I limit my study to single-chip RISCs, and explain my methods. Sections 4.2 and 4.3 then examine instruction buffers and target instruction buffers in detail. Sections 4.4 and 4.5 compare instruction buffers against target instruction buffers and draw conclusions.

4.1.1. Instruction Memory Background

High-speed instruction memories reduce instruction fetch delays by caching instructions, prefetching instructions, or both. Caching instruction memories reduce effective instruction access time (instruction fetch latency) directly by rapidly servicing fetches for recently-used instructions, and indirectly by reducing inter-chip traffic. Such traffic can cause delays by making resources unavailable for other uses, such as data references. An overview of caching can be found in [Smit82]. Prefetching instruction memories reduce effective access time by hiding part of the latency of fetching instructions.

Prefetching is effective if most prefetches bring in instructions that are executed, and if the traffic generated by the useless prefetches does not cause too many indirect delays. A discussion of prefetching can be found in [Smit78] and [Smit82].

Instruction buffers use caching and prefetching to reduce effective access time. I define an *instruction buffer* to be high-speed memory devoted to caching *all* recently executed instructions, and use the term interchangeably with *instruction cache*. Computers that use instruction buffers include: IBM System 370/ Model 158 with an IB of 32 direct-mapped, 8-byte blocks [Rau77]; the CDC 7600 with an IB (called an *instruction slave stack*) that tried to hold the last 10 and next 2 dynamic instruction words [Holg80]; and the Cray-1 with an IB of four 128-byte blocks [Lee84, Siew82]. Current commercial microprocessors with IBs include the Motorola MC68020 [Moto84] and the National NS32532 [Alpe87].

Three key papers on IBs were written by Goodman [Good83], J. Smith and Goodman [Smit83], and Agarwal et al. [Agar87]. Goodman emphasizes that in addition to reducing effective access time, instruction buffers can be effective at reducing instruction traffic, and re-introduces loading partial blocks on misses as a method for further reducing instruction traffic. I expect loading partial blocks to be important for some single-chip instruction memories, and I have incorporated loading partial blocks in the SPUR IB design. Loading partial blocks is less important, however, for cache designs where the latency for loading the first word of a miss is much larger than the delay for subsequent words (e.g., Fairchild CLIPPER) [Smit87].

J. Smith and Goodman [Smit83] use a loop model and trace-driven simulation to show that fully-associative placement with LRU replacement can lead to worse than expected performance in small instruction caches where locality is manifested predominantly as looping behavior. Consistent with their results, I find that direct-mapped instruction buffers offer comparable performance to more expensive set-associative buffers at many modest buffer sizes.

Agarwal et al. [Agar87], were the first to publish a paper examining architectural and implementation tradeoffs regarding instruction buffers on a single-chip RISC. I agree with their assertion that on-chip cache implementation concerns that affect cache hit and miss times are more important than many architectural concerns that affect only cache miss ratio. They found that cache hit time, cache size, and cache aspect ratio (the ratio of length to width of the cache implementation) are the most important implementation concerns. I found the same three concerns most important in the SPUR IB design. Agarwal et al. contrast three instruction buffer organizations: a set-associative cache with small blocks ($\leq$ 16 bytes), a fully-associative cache with larger blocks ($\geq$ 32 bytes), and a large-block cache with several sets. The final organization containing 2K bytes with four sets and 64-byte blocks was selected for MIPS-X, principally because the other two organizations could not easily be implemented to meet the above concerns. In particular, the other two designs have too great a hit time. The small block cache is too slow, because the numerous address tags cannot be placed in the datapath; the fully-associative cache is too slow, because the instruction array cannot be accessed until after the block select signal is determined by the tag array read.

Other instruction memory organizations place a greater emphasis on prefetching instructions and less emphasis on caching, which makes particularly good sense when off-chip bandwidth is plentiful. The simplest structure that supports prefetching is a *prefetch buffer* (PB). A PB is a FIFO that holds bytes sequentially forward from the current program counter. If all instructions are aligned and four bytes long, as in RISCs, a PB can be implemented as a FIFO of instruction words. Many computers have had PBs, including the IBM System/370 Model 158 [Rau77], DEC VAX-11/750 [Digi80] and DEC VAX-11/780 [Digi80].

The use of a PB always increases instruction traffic, as instructions after the end of instruction *runs* are prefetched, but never executed. An instruction *run* is a dynamic series of sequential instructions beginning with the instruction at the target of a PC-changing instruction and ending with a PC-changing instruction [Groh82]. Unconditional jumps, taken conditional branches, calls, and returns are examples of PC-changing instructions. I will loosely refer to all PC-changing instructions as *branches*. Instruction runs should not be confused with *basic blocks*. Basic blocks are found in static code, not in

dynamic instructions, and are terminated by an instruction that *may* change the PC.

CPU performance with a PB alone would be adequate if branches were rare. Since branches are not rare, however, additional steps are usually taken to reduce the effective access time of instructions at and after branch targets. *Target prefetch buffers* (TPBs) [Groh82] and *target instruction buffers* (TIBs) [Rau77] are instruction memories that hold instructions prefetched and cached, respectively, at branch targets. A target prefetch buffer holds several sequential instructions prefetched at the next predicted branch target. Machines that use target prefetch buffers include the IBM System/370 Models 195 and 168 [Holg80, Rau77]. One reason target prefetch buffers are not appropriate for single-chip RISCs is that they increase instruction traffic beyond that required by a PB alone, since prefetches must now be issued sequentially and at branch targets. In particular after a conditional branch has been detected, two instructions must be fetched per cycle for several cycles. Since it is unlikely this peak demand can be supported in a single-chip RISC, the use of a target prefetch buffer will not significantly reduce branch delay. For this reason I do not consider target prefetch buffers further.

Using a target instruction buffer (TIB) with a PB reduces effective instruction access time and instruction traffic relative to using a PB alone. A TIB contains a number of instruction-run entries, similar to IB blocks. Each entry, also a called a *block*, contains a block valid bit, an address tag and several instruction words, possibly with word-valid bits. For valid blocks, the address tag contains the address of the first instruction in an instruction run. This address tag contains more bits than that of an IB address tag, because instruction runs do not necessarily start on aligned block boundaries. If valid, the first instruction word contains the branch target instruction. Subsequent valid instruction words contain subsequent sequential instructions. The only single-chip RISC microprocessor that currently uses a TIB is the AMD Am29000[†] [Adva87].

A TIB and PB work together as follows: whenever the execution unit generates a non-sequential reference, processing on the last instruction run ceases and a TIB access is made that either hits or misses. If it hits, the TIB provides the instruction to the execution unit and tells the PB to begin prefetching after the end of the TIB block. The TIB responds to subsequent sequential fetches until the its block is exhausted; further sequential fetches are handled by the PB. On TIB miss, an off-chip fetch is made to load the instruction into the TIB and pass it to the execution unit. In parallel, the PB is reset to prefetch at the next sequential address. The PB handles all subsequent sequential fetches; these instruction are also loaded into the TIB until the TIB block is exhausted.

Prefetching and caching, in general, are discussed by Smith in [Smit78] and [Smit82]. Among many other cache aspects, the latter paper summarizes and extends the work in [Smit78] on prefetching in unified CPU caches. Smith finds that prefetching is attractive for caches with block sizes of 256-bytes and smaller if prefetch misses cost less than demand misses, prefetch operations do not impede normal cache operations, and prefetch logic does not significantly impact machine cost or cycle time. Prefetching works less well with larger blocks, because the data prefetched is less likely to be used as it further from the current reference, and a larger block must replace more currently-resident, potentially-useful data (memory pollution). Smith also finds that the only reasonable datum to prefetch in CPU caches is the block spatially after the current reference, and that, unless implementation concerns dictate otherwise, it is better to prefetch after every reference than to only prefetch after each miss. The research here differs from Smith's, because I consider prefetching of blocks containing only one or two instructions into instruction-only caches, rather than prefetching 32 bytes or more into unified caches.

Prefetching and caching of instructions, in particular, are discussed by Rau and Rossman [Rau77], Grohoski and Patel [Groh82] and Low and Rugg [Low87]. Rau and Rossman model the performance of an instruction memory hierarchy consisting of a PB in front of an IB or TIB in front of a memory. Their model recursively defines the delay for an instruction word as some maximum delay less the parts of that delay that overlap with the delays for previous instruction words. They use several instruction traces to drive a simulation of their model with the following additional assumptions: (1) instruction and data references never interfere, (2) instructions can be decoded in one cycle, and (3) the access

[†] AMD calls their TIB a *branch target cache*.

times to the PB, IB/TIB, and memory are 1, 2, and 10 cycles, respectively. They find TIBs are preferred to IBs if buffer size is small, such as 64 bytes, and if enough prefetch bandwidth is available, such as 0.8 instructions per cycle (i.e., four simultaneous prefetches of two instructions each to a 10-cycle memory). Otherwise, results show an IB is preferred to a TIB. Our results regarding when a TIB is preferred to an IB qualitatively agree with Rau and Rossmann's.

Grohoski and Patel [Groh82] model a PB and target prefetch buffer in front of a memory with and without interference from data references. Their model is based on runs of instructions that begin either at an unconditional or a conditional branch target. Their model's predictions and trace-driven simulation results closely agree for the two traces presented. Assuming 5-cycle memory, 2-cycle instruction decode and 5-cycle instruction execution, they found that increasing PB size from 0 to 1 word doubles throughput, increasing from it 1 to 2 words adds 6 percent more, and all further increases add less than 3 percent to throughput. Target prefetch buffer results for these assumptions show that the existence or size of the target prefetch buffer affects instruction traffic, but not throughput.

Low and Rugg [Low87], in a recent class project at U.C. Berkeley, examine TIBs and compare them to IBs, using trace-driven simulation with three SPUR traces and three 68000 traces. They assume perfect single-cycle sequential prefetching and examine buffer sizes from 128 to 2K bytes. They find that (1) their data predict a miss ratio for the AMD Am29000's TIB that is consistent with AMD's published data [Adva87], (2) a two-way set-associative TIB is "5 to 10 percent" better than a direct-mapped one, (3) TIB index choice is unimportant (see Section 4.2.2.1), and (4) 8-byte blocks should be preferred to 16-byte blocks. My results agree, except for (2). I find that the effective access times of two-way set-associative TIBs only 1 to 3 percent better than those of direct-mapped TIBs. I also find that many conclusions change, when IB versus TIB analysis is extended to systems without perfect single-cycle sequential prefetching.

In this chapter I examine instruction memories for reducing effective instruction access time, i.e., the delay from dereferencing instruction addresses. Instructions memories can also be used to reduce pipeline *bubbles*, resulting from not knowing the next instruction address [Holg80]. Examples of such memories are a *branch target buffer* (BTB) [Lee84, McFa86] and a *decoded instruction cache* [Ditz87, McLe82]. I do not examine such memories here, because I consider reducing pipeline bubbles to be orthogonal to reducing instruction fetch latency. Nevertheless, Figure 4-1 contrasts a BTB with a TIB and an IB, so that the three are not confused.

4.1.2. Why Limit Study to Single-Chip RISCs?

A study of instruction memory on a single-chip RISC microprocessor is merited, because RISC microprocessors are becoming more common [Adva87, Ditz87, Fuji87, Mous86], and because instruction memory design is more critical for single-chip implementations of RISC architectures than it is for traditional architectures in other technologies. I choose to restrict the scope to this study to single-chip RISCs so that I can include selected implementation assumptions, which if ignored could lead to misleading results.

For this discussion, I characterize a RISC architecture as one designed to issue an instruction every cycle [Henn84, Patt82]. A RISC architecture places more stringent demands on instruction memory than does a conventional CPU, since there are fewer cycles between instruction fetches, and its performance decays more rapidly if effective instruction access time increases. SPUR, for example, uses one to two cycles per instruction, as compared to the VAX-11/780, which uses 10.6 [Emer84]. SPUR, therefore, must fetch 0.5 to 1.0 instructions per cycle, whereas the VAX-11/780 must fetch only 0.09 instructions per cycle. If effective instruction access time increases by two tenths of a cycle, SPUR slows down 10 to 20 percent, but the VAX-11/780 slows only 2 percent.

Before the mid-1980s, the implementation of instruction memory on a single (VLSI) chip was made difficult primarily by limited chip area, and, to a lesser extent, by limited off-chip bandwidth. No microprocessors of that period devoted any significant area to instruction memory. Improved levels of integration now make including instruction memories possible and are increasing instruction memory sizes. For example, 256-byte instruction caches are used in the Motorola MC68020 [Moto84], first

A branch target buffer (BTB), target instruction buffer (TIB), and instruction buffer (IB) are easily confused. I can reduce the confusion by discussing the following branch and related instructions:

```
100         branch  TRUE, 0x180     # absolute branch
104         add     r1, r2, r3      # add

...

180         sub     r1, r2, r3      # subtract
184         sub     r4, r5, r6      # a second subtract
```

I define:

```
(a) branch instruction address = ''0x100''
(b) branch instruction = ''branch  TRUE, 0x180''
(c) branch target address = ''0x180''
(d) branch target instruction = ''sub  r1, r2, r3''
```

A BTB's purpose is to reduce pipeline bubbles, resulting from waiting for the next instruction address (program counter) to be determined. A BTB maps a branch instruction address (a) to a branch target address (c). E.g.:

```
0x100 ==> 0x180
```

A TIB's purpose is to reduce effective instruction access time (instruction fetch latency) by caching the beginning of instruction runs. A TIB maps a branch target address (c) directly to a branch target instruction (d) and indirectly to subsequent instructions. E.g.:

```
0x180 ==> ''sub  r1, r2, r3,'' ''sub  r4, r5, r6,'' ...
```

An IB's purpose is to reduce effective instruction access time by caching aligned blocks of recently-executed instructions. An IB maps an instruction address to an instruction, regardless of branch instructions. E.g.:

```
0x100 ==> ''branch  TRUE, 0x180''
0x104 ==> ''add  r1, r2, r3''
0x180 ==> ''sub  r1, r2, r3''
0x184 ==> ''sub  r4, r5, r6''
```

Since a TIB and an IB serve a different function than a BTB, either a TIB or an IB can be used after a BTB. If a TIB is used after a BTB, it can be cost-effective to combine the two into one memory that maps a branch instruction address (a) to branch target instructions (d). Such a memory has also been called a BTB.

Figure 4-1. BTB vs. TIB and IB.

shipped in 1985, and the recently-announced Motorola MC68030 [Moto86]; 512 bytes are used in the recently-announced AMD Am29000 [Adva87] and National NS32532 [Alpe87]; and 2K bytes are present on two research microprocessors [Chow87, Joup86].

VLSI chips also have more limited external bandwidth than do board-level CPUs, principally because bus width is more constrained. Instruction memories for single-chip CPUs must therefore pay more attention to minimizing instruction traffic than did their board-level predecessors. Future technological trends promise to make off-chip bandwidth, measured in instructions per cycle, more limited despite integrated circuit packaging that permits more signal pins. Paradoxically, off-chip bandwidth diminishes as the chips get faster, because intra-chip cycle times get shorter more rapidly than inter-chip communication times, making single-cycle external accesses more difficult. Consequently, the performance of future single-chip RISC instruction memories will be limited more by off-chip bandwidth than by chip area.

4.1.3. Methods

I evaluate alternative instruction memories with numerous trace-driven simulations that directly compute miss ratios and effective access times. I use trace-driven simulation for many of the same reasons that justify it in Chapter 3 (see also [Smit82] and [Smit85]). In addition, the instruction memory descriptions used here include many details of the SPUR pipeline and external cache interface. I find that it is easier to write a simulation description of these details than to develop an analytic model. Furthermore, I do not have confidence in results based on a model that ignores these details, because I have found implementation details often affect architectural trade-offs in single-chip RISC microprocessors.

Because caches here are small (≤ 8K bytes) and therefore have large miss ratios, cold-start effects and system and multiprogramming behavior are less important here than they are in large cache simulations. For this reason I ignore cold-start and system effects, and I model multiprogramming simply by flushing each cache every 30,000 references (23,800 instructions). This interval is larger than what is sometimes used to model multiprogramming in timesharing systems (e.g., 10,000 [Smit82]), but is smaller than what some researchers foresee for high-performance personal workstations with large main memories (e.g., SPUR) [Patt87]. In any case, results for these cache sizes are not sensitive to exact values of this parameter.

Like the analysis of large caches, instruction memory analysis uses miss ratio and effective access time. Large cache analysis shows that design decisions should not be made using miss ratio alone; instead the following model of effective access time, $t_{eff}(C)$, was employed:

$$t_{eff}(C) = t_{cache}(C) + m(C)*t_{memory}(C)$$

where C stands for all aspects of cache C, $m(C)$ is the miss ratio of cache C, $t_{cache}(C)$ is its hit time, and $t_{memory}(C)$ is its miss penalty.

Here I use a more complex effective access time model to take into account delays from data references and prefetches that interfere with instruction fetch misses. I assume that data references have a higher priority than instruction fetches, which have a higher priority than prefetches. A data reference initiated at the same time as an instruction fetch will cause the instruction fetch to be delayed, while a prefetch will not delay an instruction fetch. If fetches are being made to a multiple-cycle external cache, then an instruction fetch can also be delayed by data references and prefetches issued in previous cycles. Tables 4-1 and 4-2 show possible interference between two references.

Interference with One-Cycle External (Cache) Memory			
Initial Reference	Subsequent Reference		
	Data	Fetch	Prefetch
Data	no	yes	yes
Fetch	no	no	yes
Prefetch	no	no	no

Table 4-1. One-Cycle External Memory.

This table shows the possible interference between a data reference, an instruction fetch, or an instruction prefetch with subsequent references using a one-cycle external cache. Data references can delay fetches or suppress prefetches, fetches can suppress prefetches, and no other interference is possible.

In my trace-driven simulator, I model effective access time with any IB by:

Interference with Multiple-Cycle External (Cache) Memory			
Initial Reference	Subsequent Reference		
	Data	Fetch	Prefetch
Data	yes*	yes*	yes*
Fetch	yes	no	yes
Prefetch	yes*	yes	yes

Table 4-2. Multiple-Cycle External Memory.

This table shows the possible interference between a data reference, an instruction fetch, or an instruction prefetch with subsequent references using a multiple-cycle external cache. The only interference not possible, between two instruction fetch misses, cannot happen since only one instruction fetch miss can be outstanding at a time.

The interference in combinations indicated with "yes*" are modeled as "no" in Section 4.4.2, simulating a multiple-cycle external cache that is pipelined to accept a new reference every cycle.

$$t_{eff}(IB) = t_{HIT} + t_{MISS} * m(IB) + t_{BLOCKED} * m_{BLOCKED} + t_{WAITING} * m_{WAITING} + t_{WASTED} * m_{WASTED} \quad \text{Eq. 4-1.}$$

where:

t_{HIT} is the time for an IB hit,

t_{MISS} is the time for an IB miss,

$t_{BLOCKED}$ is the time lost when an IB miss can't reference the external cache due to a data reference in progress,

$t_{WAITING}$ is the time spent waiting for a prefetch in progress,

t_{WASTED} is the time wasted waiting for a prefetch in progress to complete before starting a demand fetch,

$m(IB)$ is the IB miss ratio, and

m_i's are the fractions of instruction references when a type-i event occurs.

Effective access time for the SPUR IB, which uses a single-cycle external cache and has a two-cycle miss penalty, reduces to:

$$t_{eff}(SPUR\text{-}IB) = 1 + 2*m(IB) + 1*m_{BLOCKED}. \quad \text{Eq. 4-2.}$$

The effective access time model for a TIB is slightly more complex, principally because sequential and non-sequential instructions are handled differently:

$$t_{eff}(TIB) = f_{non\text{-}seq} * (t_{TIB\text{-}HIT} + t_{TIB\text{-}MISS} * m(TIB)) + (1 - f_{non\text{-}seq}) * (t_{PB\text{-}HIT} + t_{PB\text{-}MISS} * m(PB)) + t_{BLOCKED} * m_{BLOCKED} + t_{WAITING} * m_{WAITING} + t_{WASTED} * m_{WASTED} \quad \text{Eq. 4-3.}$$

where:

$f_{non\text{-}seq}$ is the dynamic fraction of non-sequential instructions,

$t_{TIB\text{-}HIT}$ is the time for a TIB hit,

$t_{TIB\text{-}MISS}$ is the time for a TIB miss,

$t_{PB\text{-}HIT}$ is the time for a PB hit,

$t_{PB\text{-}MISS}$ is the time for a PB miss,

$t_{BLOCKED}$ is the time lost when a TIB miss can't reference the external cache due to a pending data reference,

$t_{WAITING}$ is the time spent waiting for a pending prefetch,

t_{WASTED} is the time wasted waiting for a pending prefetch to complete before starting a demand fetch,

$m(TIB)$ is the TIB miss ratio,

$m(PB)$ is the PB miss ratio, and

m_i's are the fractions of instruction references when a type-i event occurs.

As with the SPUR IB, a model of effective access time for a SPUR TIB/PB is simpler, because using a single-cycle external cache reduces interference possibilities. SPUR TIB/PB's effective access time is:

$$\begin{aligned} t_{eff}(SPUR\text{-}TIB) = {} & 1 + 2*f_{non\text{-}seq}*m(TIB) \\ & + 2*(1-f_{non\text{-}seq})*m(PB) \\ & + 1*m_{BLOCKED} + 1*m_{WAITING} \end{aligned} \qquad \text{Eq. 4-4.}$$

where t_{HIT}'s are 1 cycle and t_{MISS}'s cost 2 cycles. $m_{waiting}$ is non-zero, because the SPUR TIB/PB uses bypassing to reduce its miss ratio (to be described).

Considerable trace-driven simulation was done during the design of the SPUR IB using traces of C programs from RISC II and Franz Lisp programs from VAX-11. The VAX-11 traces provided only coarse IB performance estimates, since the VAX-11 and SPUR instruction sets differ considerably. SPUR traces and, in particular, SPUR Lisp traces were not used because the software was not ready until after the IB had been partially implemented. Retrospective analysis indicates that RISC C programs produced more optimistic IB performance results than SPUR Lisp programs have, since the former have fewer loads and stores (10-15 vs. 20-30 percent of instructions) and generally better locality in small instruction caches.

For this reason, all results in this chapter use traces of SPUR Lisp programs. In particular I use the three large programs that currently run on SPUR's architectural simulator [Tayl87]. They are:

(1) *Slc*, the SPUR LISP compiler [Zorn87], based on the SPICE LISP compiler [Whol85], compiling part of itself;

(2) *Rsim*, a circuit simulator simulating a counter [Term83];

(3) *Weaver*, a production system written on top of OPS5 for VLSI chip routing [Joob85].

For each of these programs, I plotted the dynamic miss ratio versus time to select two 500K-instruction trace samples that have different, but somewhat pessimistic behavior. Figure 4-2 shows the miss ratios of the six samples and their arithmetic average. Since miss ratio variation across the trace samples is small, subsequent results are based on miss ratios and the effective access times for a composite trace, formed by concatenating the six trace samples. Since the length of samples is the same, the miss ratios and effective access times for the composite trace are equal to the arithmetic averages of miss ratios and the effective access times from the individual samples. I do not use the well-known Gabriel benchmarks [Gabr85] for this analysis, because all their code sizes are smaller than some instruction buffer sizes I examine.

Finally a word regarding how effective access time affects cycles per instruction is warranted. While absolute changes in $t_{eff}(SPUR\text{-}IB)$ translate directly into absolute changes in cycles per instruction, relative changes in effective access time translate into smaller relative changes in cycles per instruction, because cycles per instruction is larger due to several factors unrelated to instruction memory. These additional factors dilute the relative impact of changes in effective access time on cycles per instruction. Reducing $t_{eff}(SPUR\text{-}IB)$ from 1.3 to 1.2 (-7.7 percent), in SPUR for example, only reduces cycles per instruction by 5.9 percent, since Gibson reports that external cache misses, two-cycle stores, and several minor factors cost 0.4 cycles per instruction regardless of instruction memory performance [Gibs87]. Nevertheless, relative changes in $t_{eff}(SPUR\text{-}IB)$ can be used as a slightly exaggerated estimate of relative changes in cycles per instruction.

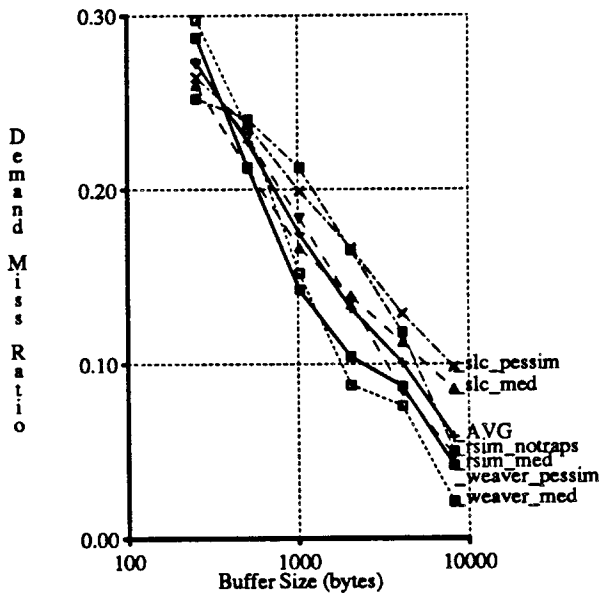


Figure 4-2. Trace Samples vs. Arithmetic Average.

The figure shows demand miss ratio versus IB size for six trace samples and their arithmetic average ("AVG"). The arithmetic average is the individual miss ratios is equal to the miss ratio of a system where each sample ran equally frequently.

4.2. Instruction Buffers

In this section I first describe the SPUR IB to document our design and a establish it as a design operating point. I then study alternate IB designs about this operating point. I find the effective access time of the 512-byte SPUR IB to be 1.51 cycles per instruction (with 1.0 being ideal) and the most cost-effective improvement to be connecting the IB to the external cache via a 64-bit data bus rather than the 32-bit bus currently used. This change reduces effective access time by 12 percent at a cost of 24 additional CPU pins[†] and the design time for a separate instruction array. Alternatively, effective access time can be reduced by 7.7 percent by doubling the IB size to 1K-bytes. Our results indicate that incorporating both changes and increasing the block size to 64 bytes produces a cost-effective "improved SPUR IB" with an effective access time of 1.21 cycles per instruction (20 percent better than the current SPUR IB).

4.2.1. SPUR IB Architecture and Implementation

The SPUR IB architecture, which evolved from February 1984 to March 1985, is shown in Figure 4-3 [Dunc86, Katz85]. The unconventional feature of the SPUR IB is a valid bit associated with each instruction word in the IB so that any subset of these words may be valid. The SPUR IB uses this flexibility to reduce demand miss time by loading only the fetched instruction rather than the entire block, and to permit instruction prefetching to load the rest of a block in parallel with subsequent instruction fetches. Improving VLSI cache performance by loading partial blocks on misses is described in [Good83] and [Hill84]. Prefetching to fill up other parts of a block is introduced in [Hill84].

The SPUR IB operates in three modes: *disabled, enabled-without-prefetching* and *enabled-with-prefetching*. The mode used is determined by the state of two bits in the KPSW (Kernel Processor Status Word). In disabled mode, the IB accepts an address from the execution unit, passes it to the external cache, receives the instruction from the external cache, and passes it to the execution unit

[†] While the IB uses only 32 data bus pins, SPUR's data bus is 40 bits wide to support tagged Lisp data.

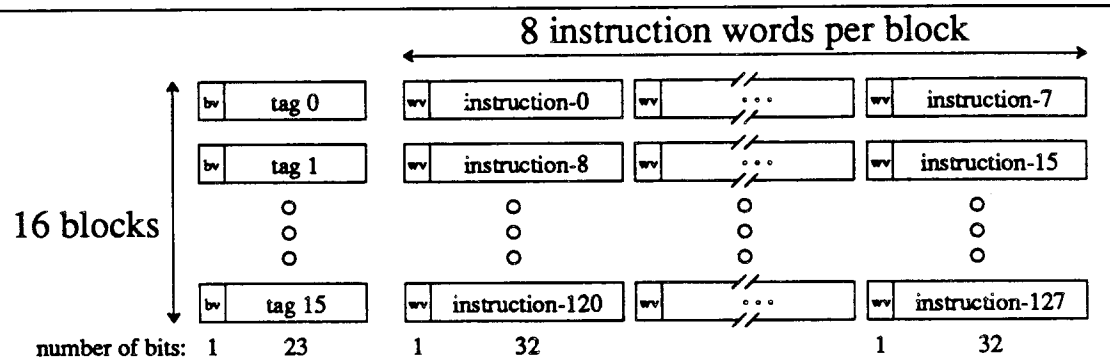


Figure 4-3. SPUR Instruction Buffer Architecture.

The SPUR instruction buffer (IB) is a 512-byte (128-instruction) instruction cache, divided into 16 direct-mapped blocks. Each block contains a *block-valid* bit (labeled *bv*), a 23-bit address tag, eight *word-valid* bits (labeled *wv*) and eight 32-bit instructions words. The block-valid bit is set to indicate a valid address tag. The address tag holds the most significant 23 bits of a 32-bit address. Each word-valid bit indicates whether the corresponding instruction word is valid. Associating valid-bits with each instruction word allows any sub-set of the instruction words in a block to be valid.

without using the IB tag or instruction memory. Disabled mode is used for initial chip testing, and for allowing chips with *stuck-at*-type errors in the IB tag or instruction memory to function correctly, albeit more slowly.

In enabled-without-prefetching mode, the IB may cache instructions, but is not allowed to initiate any prefetches. An instruction access to the IB may cause a *hit* (tag match, block-valid bit and word-valid bit set), *block-miss* (tag mis-match or block-valid bit not set), or *word-miss* (tag match, block-valid bit set, but word-valid bit not set). On a hit, the instruction is immediately returned to the execution unit, and the state of the IB is unaffected. On a block-miss, the instruction is fetched from the external cache and loaded into the corresponding instruction word, the tag is set to the instruction's address, the block-valid bit and corresponding word-valid bit are set, and all other word-valid bits are reset. On a word-miss, the instruction is fetched from the external cache, loaded into the corresponding instruction word and the corresponding word-valid bit is set.

The normal operating mode of the IB in a working system is enabled-with-prefetching. In this mode the IB caches instructions in the manner just described, but adds to this a prefetch request to SPUR's single-cycle external cache on every cycle where that cache would otherwise be idle. These prefetches are "free" is the sense that they never interfere with other potentially more-useful external cache accesses. After an IB miss, prefetches are made to subsequent words within the block until another demand miss occurs. Instead of terminating prefetching at the end of the block or when all instructions in the block are loaded (by wrapping around), the SPUR IB never terminates prefetching; instead it prefetches instructions in the block of the last demand miss over and over again, until another demand miss occurs, signaling a new useful prefetch location. The external cache may abort prefetches without providing data by asserting the "data-not-available" signal. The aborting of prefetches is useful for guaranteeing that a prefetch that misses in the external cache cannot cause protection exceptions, page faults or system bus traffic.

The IB architecture also includes two systems operations, *IB-reset* and *IB-flush*, for abandoning fetches or prefetches and for invalidating the contents of the IB, respectively. I omit discussing these details, as they are not germane to the SPUR IB's performance.

The appendix at the end of this chapter, Section 4.6, presents details of the SPUR IB's implementation, including diagrams of its control finite state machines (FSMs) and its phase-by-phase behavior on hits and misses. Additional information on this subject and the circuits used is provided in [Dunc86]. The principal assumptions from the IBs implementation, rather than architecture, used in the IB analysis of Section 4.2.2 are that (1) instruction misses cost two additional cycles -- a one cycle

external cache access plus one cycle on-chip, (2) instruction prefetches take one cycle for an external cache access, and (3) data references can block instruction fetch misses or suppress prefetches for one cycle.

The implementation concerns that drove the SPUR IB design were that it have a one-cycle (or less) access time, and that it be as easy as possible to build. The one-cycle access time was required to enable the effective access of the IB to be less than that achieved by fetching instructions directly from the external cache. We wanted the SPUR IB, and all other SPUR system components, to be easy to build to facilitate the design and implementation of a complete system in four years. The project's plan was to build a system with straightforward components, and then to retrospectively study less conservative component designs by varying parameters about operating points determined by the actual system components.

The requirement for easy-to-build components affected the IB design in two general ways. First, whenever possible, we tried to re-use cells that had already been designed and tested. Second, we tried to keep simple the requirements for the IB control logic.

The most important instance of re-using cells in the IB is implementing the IB instruction array with the memory cells of the register file. The advantages of doing this were that another memory cell did not have to be designed to meet the speed and clocking requirements of SPUR, the buffer and drivers of the register file could be used with little or no adaptation, and execution unit datapath logic, such as the adder cells, could be re-used in the IB datapath, since the pitch of both datapaths would be the same. Using the register file memory cells in the instruction array, however, reduces the potential capacity of the IB and restricts its aspect ratio. IB capacity is reduced because the register file cell: (1) is optimized for the register file, which is larger than the IB's instruction array (138 by 40 vs. 128 by 32), (2) supports dual-port read, which is not used by the IB, and (3) is fully-static. Lee estimates that optimizing the cell for the instruction array and removing one of the two metal word lines in each cell would reduce the IB memory array size by 20% [Lee87]. A much larger reduction is possible if a 3-transistor dynamic cell is used in the IB array[†]. Lee estimates that the 3-T dynamic cell would be less than one-quarter the size of the 6-T static cell, principally because no well boundary need be present in the dynamic cell without pull-up transistors. This size reduction would permit IB capacity increase by a factor of two to four. The aspect ratio of the register file cells constrains the IB array to be 32 bits wide. This restriction makes it difficult to either build a set-associative IB or to load more than 32 bits on a miss or prefetch.

The most important instance of simplifying the IB logic was decision to partition it into two FSMs that control fetches and prefetches, respectively, and to allow the two FSMs to interact only on demand misses (see Section 4.6). This decoupling simplified control logic, but decreases IB performance, since the fetch FSM is unaware of what is being prefetched. Consequently, time can be wasted when the fetch FSM issues an external cache fetch for an instruction reference that had missed in the IB but has been subsequently loaded by a prefetch. The principal drawback of decoupled control logic can be reduced using bypass logic, but only at the cost of the complexity arising from more interaction between the two FSMs.

4.2.2. IB Evaluation

This section examines the performance of IBs like the SPUR IB. I begin by varying three architectural parameters: buffer size, associativity, and block size; then I examine the effects of changing prefetch algorithms and the bandwidth available for misses and prefetching; next I look at schemes for

[†] It is more reasonable to use dynamic memory in the IB than in the register file for two reasons. First, the IB can be disabled during initial chip testing when one may want to run parts slowly. The chip cannot be tested at a slow rate if register file cells lose their values. Second, dynamic cells in the IB array may be invalidated, rather than refreshed, since copies of all instructions still reside in main memory. Thus, the simplest IB "refresh" circuitry is a counter that flushes the IB every 2 milliseconds. Normal memory-refresh circuits, on the other hand, are necessary for dynamic register file cells.

reducing the IB miss penalty; and finally I propose an improved IB. The miss ratios and effective access times discussed in the section are presented in Tables 4-3 and 4-4. I calculate effective access times from miss ratios using the equations given in Section 4.1.3, assuming a constant cycle time of 1.0. In general, effective access times are slightly greater than one plus twice the miss ratio.

4.2.2.1. IB Size, Associativity, and Block Size

The architectural feature that has the greatest impact on IB miss ratio is IB size. Figure 4-4 shows miss ratios and effective access times for IBs of size 256 to 8K bytes. Results show a linear reduction of the miss ratio of 0.043 for each doubling of direct-mapped IB size within this range. This trend does not continue for larger sizes; if it did, the miss ratio would be zero for a 32K-byte IB. Nevertheless, it underscores that for small IB sizes, increasing IB size has a large effect on IB miss ratio. Miss ratios are reasonable, since they are consistent with my extension of Smith's design target miss ratios for direct-mapped instructions caches, given in Chapter 3.

The degree to which lower miss ratios for larger IBs translate into smaller effective access times, however, is related to how rapidly larger IB implementations can be accessed. The right plot in Figure 4-4 shows effective access times for IBs of various sizes, assuming a constant CPU cycle time. The effective access times for 512 and 1K-byte IBs, for example, are 1.51 and 1.39 cycles. Thus, doubling IB size decreases effective access time by 7.7 percent if cycle time is unaffected. Eventually, however, the IB will be on the critical path that determines the cycle time, and the IB size cannot be increased without increasing the cycle time. Duncombe [Dunc86] shows, for example, that a naive implementation of a 1K-byte SPUR IB, using the unmodified register file cell, could have a 14 percent slower critical signal delay. This could lead to, at most, a 7 percent increase in SPUR cycle time. If this happened, the effective access time improvement from doubling IB size would be only 1.3 percent.

Table 4-4 also shows that changing IB associativity is of minor importance[†]. While doubling the SPUR IB's associativity decreases its miss ratio from 0.2275 to 0.2151, this change yields only a 1.6 percent (relative) reduction in effective access time. The magnitude of this change is small enough to be easily overwhelmed by access time differences between each of the implementations with different associativities. Duncombe [Dunc86] shows that several implementation changes, for example, in the memory array aspect ratio, can change critical single delays by over five percent. At 2K and 4K bytes, doubling the associativity to two-way improves effective access time by a non-negligible 3.1 and 4.1 percent. Nevertheless, a small access time increase can wipe out this gain.

Results also show 16-byte (four instruction) blocks should be avoided, because they yield effective access times 2 to 10 percent greater than the other block sizes examined. Block sizes of 32, 64 and 128 bytes yield comparable performance with a singleword (32-bit) bus, and effective access times decrease slightly for increasing block size when a doubleword (64-bit) bus is used (with doubleword sub-blocks). Large blocks work well, because the latency for loading a block on a miss does not increase for larger blocks as additional words are loaded in parallel with subsequent instruction fetches. For a similar reason, Smith finds 64-byte blocks optimal for 128 to 2K-byte instruction caches when the cost of loading subsequent words is almost free (1/40-th the cost of loading the first word) [Smit87]. Since 32, 64 and 128-byte yield similar performance here, IB block size should be selected by examining implementation considerations. One reason to prefer larger blocks is that they require fewer address tags, which saves chip area and can reduce IB access time.

[†] Results use random replacement since its performance is comparable to LRU replacement for small instruction caches [Smit83] and it is slightly easier to implement than LRU replacement since nothing must be written on an IB hit. LRU replacement, on the other hand, requires an LRU bit to be written in a two-way set-associative IB, and even more complexity for associativities greater than two. The miss ratio for a 512-byte two-way set-associative IB with LRU replacement is 0.2204 (not shown), slightly larger than the miss ratio with random replacement. This corroborates J. Smith and Goodman's findings that random replacement is superior to LRU replacement in small instruction caches where loop behavior dominates.

Distinguishing Attribute(s)	Size (bytes)					
	256	512	1024	2048	4096	8192
SPUR IB	0.2731	0.2276	0.1748	0.1321	0.1008	0.0581
Associativity						
1	0.2731	0.2276	0.1748	0.1321	0.1008	0.0581
2	0.2661	0.2151	0.1607	0.1123	0.0787	0.0511
4	0.2674	0.2091	0.1446	0.1074	0.0731	0.0464
8	0.2675	0.2096	0.1422	0.1061	0.0705	0.0445
Block Size (bytes)						
16	0.3075	0.2552	0.1994	0.1525	0.1160	0.0675
32	0.2731	0.2276	0.1748	0.1321	0.1008	0.0581
64	0.2617	0.2235	0.1705	0.1293	0.1001	0.0573
128	0.2560	0.2264	0.1786	0.1334	0.1034	0.0608
Associativity w/ Doubleword Bus						
1	0.1840	0.1495	0.1138	0.0854	0.0645	0.0362
2	0.1796	0.1420	0.1036	0.0704	0.0489	0.0316
4	0.1803	0.1379	0.0918	0.0667	0.0451	0.0284
8	0.1805	0.1384	0.0899	0.0657	0.0434	0.0272
Block Size w/ Doubleword Bus						
16	0.2552	0.2103	0.1643	0.1263	0.0957	0.0542
32	0.1840	0.1495	0.1138	0.0854	0.0645	0.0362
64	0.1589	0.1321	0.0999	0.0755	0.0587	0.0307
128	0.1398	0.1192	0.0887	0.0664	0.0509	0.0284
Prefetch Algorithms w/ One Singleword Bus						
demand	0.8232	0.6971	0.5530	0.4338	0.3352	0.1923
w/a-miss	0.4438	0.3737	0.2964	0.2317	0.1781	0.1023
MIPS-X	0.4160	0.3540	0.2827	0.2229	0.1732	0.0992
remainder	0.2690	0.2244	0.1731	0.1327	0.1019	0.0588
wrap-around	0.2675	0.2232	0.1719	0.1314	0.1008	0.0583
SPUR	0.2731	0.2276	0.1748	0.1321	0.1008	0.0581
always	0.2177	0.1800	0.1370	0.1055	0.0811	0.0468
One Doubleword Bus						
demand	0.4559	0.3819	0.2994	0.2326	0.1787	0.1028
w/a-miss	0.2692	0.2232	0.1728	0.1328	0.1015	0.0578
MIPS-X	0.3896	0.3318	0.2651	0.2089	0.1627	0.0921
remainder	0.2047	0.1678	0.1276	0.0972	0.0740	0.0419
wrap-around	0.2028	0.1659	0.1256	0.0951	0.0723	0.0411
SPUR	0.1840	0.1495	0.1138	0.0854	0.0645	0.0362
always	0.1383	0.1102	0.0808	0.0592	0.0450	0.0256
Two Singleword Buses						
demand	0.8232	0.6971	0.5530	0.4338	0.3352	0.1923
w/a-miss	0.4438	0.3737	0.2964	0.2317	0.1781	0.1023
MIPS-X	0.4160	0.3540	0.2827	0.2229	0.1732	0.0992
remainder	0.1927	0.1576	0.1192	0.0909	0.0686	0.0389
wrap-around	0.1910	0.1561	0.1177	0.0893	0.0672	0.0382
SPUR	0.1988	0.1631	0.1237	0.0926	0.0693	0.0399
always	0.1118	0.0871	0.0613	0.0451	0.0335	0.0187
Two Doubleword Buses						
demand	0.4559	0.3819	0.2994	0.2326	0.1787	0.1028
w/a-miss	0.2692	0.2232	0.1728	0.1328	0.1015	0.0578
MIPS-X	0.3896	0.3318	0.2651	0.2089	0.1627	0.0921
remainder	0.1867	0.1522	0.1151	0.0877	0.0663	0.0372
wrap-around	0.1846	0.1501	0.1129	0.0855	0.0645	0.0362
SPUR	0.1777	0.1441	0.1095	0.0824	0.0619	0.0348
always	0.1040	0.0799	0.0563	0.0410	0.0306	0.0167

Table 4-3. IB Miss Ratios.
Parameters not listed match those of the SPUR IB, described in Section 4.2.1.

4.2.2.2. Off-Chip Bandwidth and Prefetching

Using more expensive external bus and cache structure will improve also the performance of the SPUR IB by providing increased off-chip bandwidth. SPUR uses the least expensive alternative,

Distinguishing Attribute(s)	Size (bytes)					
	256	512	1024	2048	4096	8192
SPUR IB	1.6139	1.5116	1.3949	1.2991	1.2295	1.1331
Associativity						
1	1.6139	1.5116	1.3949	1.2991	1.2295	1.1331
2	1.5977	1.4870	1.3650	1.2583	1.1817	1.1176
4	1.6005	1.4736	1.3312	1.2472	1.1684	1.1065
8	1.6009	1.4744	1.3260	1.2444	1.1623	1.1023
Block Size (bytes)						
16	1.6919	1.5752	1.4521	1.3458	1.2637	1.1547
32	1.6139	1.5116	1.3949	1.2991	1.2295	1.1331
64	1.5863	1.5005	1.3846	1.2926	1.2273	1.1307
128	1.5730	1.5064	1.4013	1.2998	1.2334	1.1381
Associativity w/ Doubleword Bus						
1	1.4022	1.3270	1.2499	1.1879	1.1425	1.0809
2	1.3921	1.3128	1.2286	1.1573	1.1095	1.0709
4	1.3936	1.3035	1.2040	1.1487	1.1008	1.0635
8	1.3941	1.3043	1.1999	1.1466	1.0971	1.0609
Block Size w/ Doubleword Bus						
16	1.5626	1.4648	1.3644	1.2805	1.2131	1.1220
32	1.4022	1.3270	1.2499	1.1879	1.1425	1.0809
64	1.3421	1.2846	1.2159	1.1634	1.1271	1.0672
128	1.2990	1.2547	1.1895	1.1412	1.1086	1.0610
Prefetch Algorithms w/ One Singleword Bus						
demand	2.8628	2.5792	2.2577	1.9866	1.7658	1.4419
w/a-miss	1.9993	1.8436	1.6719	1.5266	1.4066	1.2346
MIPS-X	1.9388	1.8003	1.6415	1.5058	1.3947	1.2273
remainder	1.6059	1.5059	1.3921	1.3009	1.2322	1.1349
wrap-around	1.6029	1.5035	1.3897	1.2984	1.2299	1.1338
SPUR	1.6139	1.5116	1.3949	1.2991	1.2295	1.1331
always	1.4892	1.4064	1.3116	1.2402	1.1854	1.1076
One Doubleword Bus						
demand	2.0223	1.8590	1.6771	1.5274	1.4071	1.2353
w/a-miss	1.5911	1.4911	1.3818	1.2942	1.2257	1.1295
MIPS-X	1.8858	1.7552	1.6059	1.4778	1.3738	1.2130
remainder	1.4490	1.3684	1.2814	1.2147	1.1641	1.0939
wrap-around	1.4449	1.3644	1.2770	1.2102	1.1606	1.0919
SPUR	1.4022	1.3270	1.2499	1.1879	1.1425	1.0809
always	1.2961	1.2372	1.1749	1.1289	1.0988	1.0561
Two Singleword Buses						
demand	2.6464	2.3941	2.1060	1.8676	1.6705	1.3847
w/a-miss	1.8875	1.7475	1.5928	1.4633	1.3561	1.2046
MIPS-X	1.8320	1.7080	1.5655	1.4458	1.3463	1.1984
remainder	1.3853	1.3151	1.2385	1.1817	1.1372	1.0778
wrap-around	1.3819	1.3122	1.2355	1.1786	1.1344	1.0764
SPUR	1.3976	1.3262	1.2473	1.1852	1.1386	1.0798
always	1.2235	1.1742	1.1225	1.0903	1.0670	1.0373
Two Doubleword Buses						
demand	1.9117	1.7638	1.5988	1.4652	1.3574	1.2056
w/a-miss	1.5384	1.4465	1.3456	1.2657	1.2030	1.1155
MIPS-X	1.7793	1.6637	1.5302	1.4177	1.3254	1.1842
remainder	1.3735	1.3044	1.2303	1.1755	1.1326	1.0744
wrap-around	1.3692	1.3001	1.2259	1.1710	1.1290	1.0725
SPUR	1.3554	1.2882	1.2190	1.1647	1.1237	1.0696
always	1.2080	1.1599	1.1126	1.0819	1.0613	1.0334

Table 4-4. IB Effective Access Times.
Parameters not listed match those of the SPUR IB, described in Section 4.2.1.

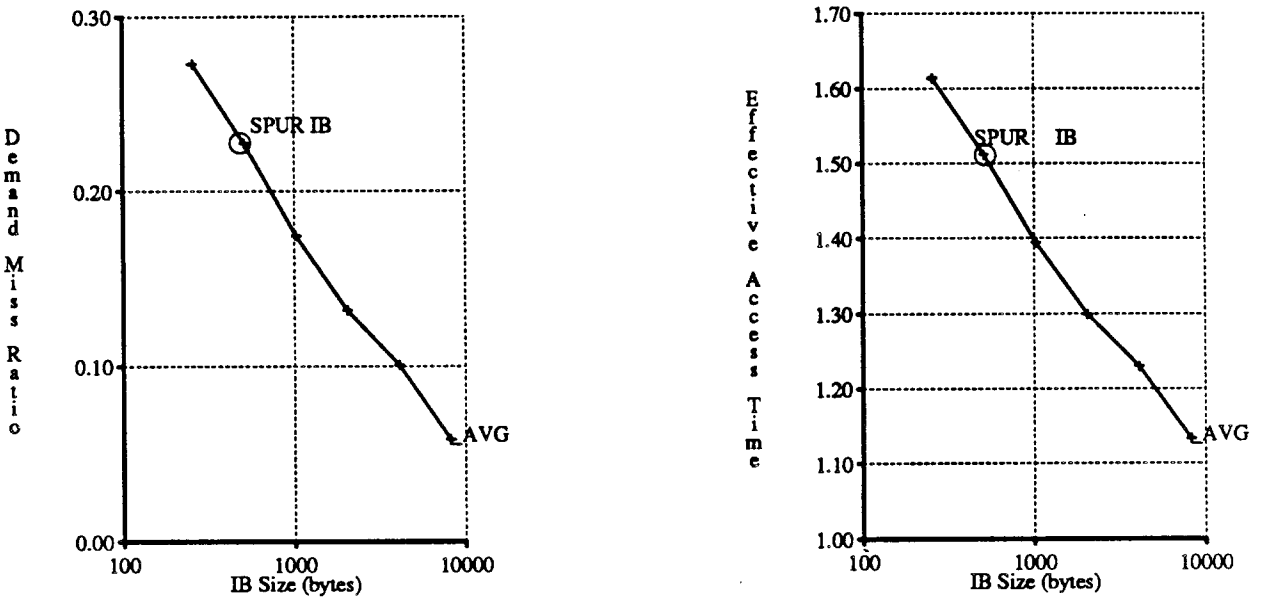


Figure 4-4. SPUR IB Performance.

This figure shows demand miss ratio (left) and effective access times (right) versus IB size. Effective access time results assume a 1-cycle hit, 2-cycle miss penalty and a constant cycle time regardless of IB size. Thus, the effective access times for large IBs may be artificially low. The size of the SPUR IB is 512 bytes.

Results show miss ratios decrease by 0.043 per doubling in cache size. If hit and miss times are not increased, this translates into 6.4 to 9.6 relative decrease in effective access time. Note that effective access times are greater than one plus twice the miss ratio, because of cycles lost when instruction fetch misses are blocked from the external cache by pending data references.

namely, it transfers information from and to the external unified cache via one singleword bus. Other alternatives are to use one doubleword bus and cache, separate singleword buses and caches for instructions and data, and separate doubleword buses.

Performance improves markedly as the size and number of buses is increased. The use of one singleword bus yields the worst performance, because a single data reference can cause sequential prefetching to fail to keep up with the execution unit. Using one doubleword bus not only allows the instruction prefetcher to keep up but also allows it to get ahead and capture short forward jumps in the best case. However, collisions between instruction and data references are still possible. The use of two buses, either singleword or doubleword, eliminates these collisions altogether. Two doubleword buses perform slightly better than two singleword buses by capturing some short forward jumps.

The implementation cost of these alternatives rises as performance improves. One singleword bus requires 64 CPU pins (32 address pins plus 32 data pins[†]) and a singleword-wide external cache. One doubleword bus uses 96 CPU pins and requires a doubleword-wide external cache. Two singleword buses require 128 CPU pins and two singleword-wide external caches. Two doubleword buses require 192 CPU pins and two doubleword-wide external caches. These costs can be changed by sharing pins between address and data buses, or by pipelining a single external cache to handle two requests per cycle. Both optimizations add complexity, however, and can increase machine cycle time.

The proper number and size of buses and external caches is determined by trading off performance benefits with implementation costs. Figure 4-5 shows the miss ratios and effective access times of the four alternatives for the SPUR IB at difference sizes. Results show that one singleword bus performs worse than the other three alternatives, which perform similarly. At 512 bytes, for example, the

[†] SPUR uses an additional eight data pins to transfer 40-bit tagged Lisp data.

effective access time with one singleword bus is 12 to 14 percent worse than the other three configurations. This difference decreases for larger IBs, since the number of collisions decreases as the miss ratio decreases. At 8K-bytes, for example, the difference is only 5 to 6 percent.

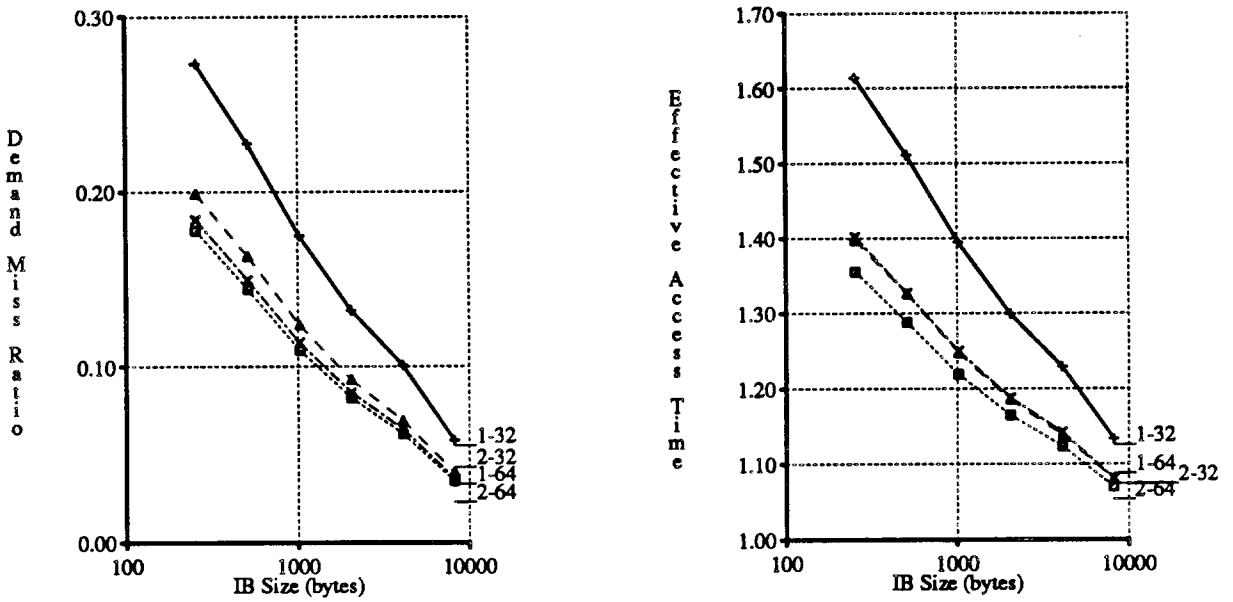


Figure 4-5. Vary External Buses.

This figure shows demand miss ratio (left) and effective access time (right) versus IB size for one or two external buses (and caches) of singleword (32-bit) or doubleword width (64-bit). SPUR uses a shared singleword bus, because it requires the least amount of external hardware and CPU pins. The other three alternatives, however, have better performance. At 512 bytes, for example, the effective access times of a doubleword bus, two singleword buses and two doubleword buses are less by 10, 12 and 14 percent (relative).

Another parameter that affects cost and performance of the IB is the prefetch algorithm. The complexity from implementing the prefetch algorithms defined in Table 4-5 manifests itself in the number of accesses to the instruction and tag arrays required per cycle, and in additional hardware, such as adders for calculating prefetch addresses. If instructions are loaded only on *demand* misses, the instruction and tag arrays must support one read or one write per cycle, and no prefetch hardware is required.

*W/a-miss-*prefetching (wrap-around-prefetching on misses only) and *MIPS-X-*prefetching (defined below) are the simplest to implement. Neither requires additional bandwidth into the instruction or tag arrays if a demand miss stalls the pipeline for two cycles, but the external cache processes requests in one cycle. Instead the prefetcher uses these arrays only in the cycle when a demand miss is being completed and the arrays would otherwise be idle. W/a-miss-prefetching requires a modulo-8 (3-bit) incrementer to calculate the prefetch address during the cycle when a demand miss is accessing the external cache. MIPS-X-prefetch, on the other hand, does not require any special adder to calculate the prefetch address, because it prefetches at the *actual* next PC, which is calculated by the MIPS-X execution unit while the IB is processing the demand miss. That PC never depends on the instruction currently missing, because MIPS-X uses delayed branches. A MIPS-X prefetch, however, can cause the replacement of any block in the IB, including the one containing the instruction just fetched. Agarwal, et al., found the complexity of prefetch replacements to be minor [Agar87].

*SPUR-*prefetching requires more implementation support. The most important additional requirement for SPUR-prefetching is an instruction array that supports a read and write each cycle. The write is used to store prefetches into the instruction array. Additional tag-array bandwidth is not required since prefetches do not cross block boundaries. The next prefetch address is calculated with a register that is reset on each demand miss, and a modulo-8 (3-bit) incrementer. One innovation of SPUR

Prefetch Algorithm	Description
Instruction Array Read *or* Write Per Cycle	
demand	Demand misses only.
w/a-miss	After each instruction miss, do a wrap-around-prefetch (see below).
MIPS-X	After each instruction miss, prefetch the sub-block of the *actual* next instruction. Since MIPS-X uses delayed branches, the address of the next instruction can be calculated while the current miss is being serviced.
Instruction Array Read *and* Write Per Cycle	
SPUR	On each idle external cache cycle, prefetch the next sub-block in the block of last demand miss. The *next* sub-block is the next sequential sub-block or the first sub-block in the block after the sub-block of the last prefetch or demand miss. Unlike the other algorithms whose prefetch address is a function of the last reference, SPUR's prefetch address depends on the address of the last demand *miss*.
remainder	On each instruction reference not to the final sub-block of a block, prefetch the next sequential sub-block; otherwise, do not prefetch.
wrap-around	On each instruction reference not to the final sub-block of a block, prefetch the next sequential sub-block; otherwise, wrap-around and prefetch the first sub-block of the same block.
always	On each instruction reference, prefetch the next sequential sub-block, even if it maps into the next block frame. This algorithm may require multiple tag array accesses per cycle.

Table 4-5. Prefetch Algorithms.

This figure describes various prefetch algorithms, listed roughly in ascending order of implementation complexity. Algorithms in the first group require that the IB instruction array support only one access, a read or a write, per cycle. Algorithms in the second group require that the instruction array support both a read and a write per cycle.

It is easy to confuse MIPS-X-prefetching with w/a-miss-prefetching and SPUR-prefetching with wrap-around-prefetching. Both w/a-miss- and MIPS-X-prefetching prefetch only after demand misses. W/a-miss-prefetching prefetches at an address that is a function of the last instruction, while MIPS-X-prefetching prefetches at the address of the *actual* next instruction. Wrap-around-prefetching prefetches after each reference at an address that is a function of the last instruction, while SPUR-prefetching prefetches each idle cache cycle at an address that is a function of the last *demand miss*.

prefetching is that the prefetch address is a function of the last demand *miss* rather than the last reference. Doing this simplies control logic by permitting prefetching to be controlled with a separate FSM that interacts with the main FSM only during demand misses.

The implementation complexity of *wrap-around*-prefetching and *remainder*-prefetching is similar to that of SPUR-prefetching. As with SPUR-prefetching, only a 3-bit incrementer is needed to calculate the prefetch address. This incrementer, however, must operate in less than a phase rather than in an entire cycle, because the next prefetch address for these schemes is a function of the address of the last reference, rather than the address of the last miss. Care must be used to insure that including an increment before each prefetch does not increase the CPU's cycle time.

Always-prefetch is more complex to implement than the other five alternatives, because two tag array accesses per cycle may be needed, and a full 30-bit incrementer is required to calculate the prefetch address. The first tag array access is for fetches and misses; the second is for prefetches. The other prefetch algorithms do not need the second tag array access, because their prefetches never cross block boundaries. The second tag array read may be done as a side-effect of the first, e.g., a read of tag n also reads tag $n+1$, and need not be done on every reference if repeated requests for the same tag are filtered. In any case, the tag array or the logic surrounding it must be more sophisticated than is required for the other prefetch algorithms. The logic to compute the prefetch address is also more complicated for always-prefetch. A full 30-bit incrementer must operate on the instruction fetch address in less than one phase. The timing of the SPUR CPU would have to be altered to permit this.

Figure 4-6 shows the various prefetch algorithms with the SPUR configuration of one singleword bus. Prefetch algorithm performance falls into four equivalence classes, which in order of decreasing effective access time are: demand-fetch; MIPS-X- and w/a-miss-prefetch; SPUR-, wrap-around-, and remainder-prefetch; and always-prefetch. At 512 bytes, for example, the effective access time for demand-fetch, MIPS-X-prefetch and always-prefetch relative to that with SPUR-prefetching is 70 percent greater, 19 percent greater and 7 percent less. The difference between the algorithms diminishes as cache size in increases. At 8K bytes, the relative differences have reduced to 27, 8.3 and -2.3 percent.

MIPS-X-prefetch and SPUR-prefetch offer the best cost-performance. If the instruction array can support only a read *or* a write per cycle, only demand-fetch, w/a-miss-prefetch and MIPS-X-prefetch are possible. MIPS-X-prefetch yields much better performance than demand-fetch for modest additional cost and slightly better performance than w/a-miss-prefetching at comparable cost. If a read *and* a write are possible, SPUR-prefetching yields performance similar to wrap-around-prefetch and remainder-prefetch, and is slightly easier to implement, since the next prefetch address is a function of the last miss rather than the last reference. SPUR-prefetching yields performance a little worse than always-prefetch, but is significantly simpler to implement.

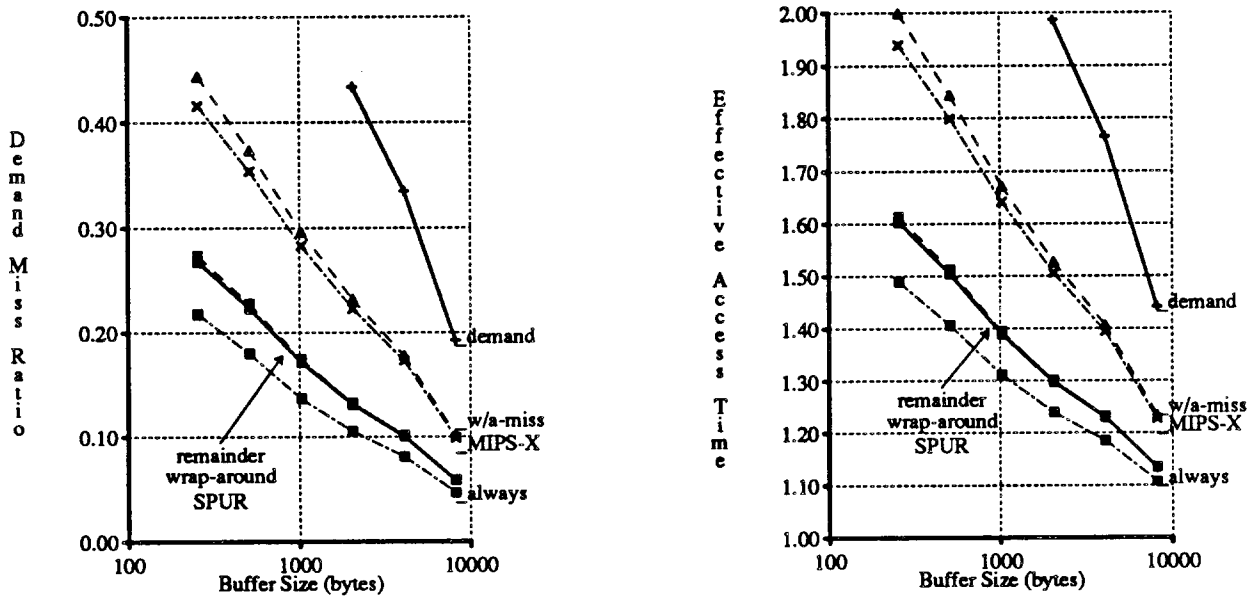


Figure 4-6. Vary Prefetch Algorithm.

This figure shows demand miss ratio (left) and effective access time (right) versus IB size for various prefetch algorithms.

The use of a doubleword bus improves the performance of all prefetch algorithms, but by non-uniform amounts (see Figure 4-7). With a doubleword bus, w/a-miss-prefetching, rather than MIPS-X-prefetching, is preferred for instruction arrays that can only be read or written each cycle. Demand-

fetching benefits the most from doubling the bus width, because a demand miss often brings in the next instruction executed. MIPS-X-prefetching benefits the least, since the additional instruction brought in on a miss is often the same one that MIPS-X-prefetching prefetches. For example, a miss to word address 0 brings in instructions at address 0 and 1. If the actual next PC is 1, MIPS-X-prefetch will redundantly prefetch an aligned doubleword at word address 1, i.e., the instructions at addresses 0 and 1. Therefore, in the common sequential case, the MIPS-X-prefetch is not useful. W/a-miss-prefetching performs better, because it does a useful prefetch in the common sequential case. A marginal performance improvement can be achieved by doing a w/a-miss-prefetch if the execution unit indicates that the next PC is sequential or a MIPS-X-prefetch if not. The implementation complexity for this, comparable to the union of the implementation complexities of w/a-miss- and MIPS-X-prefetch, however, is not justified, especially for large IBs.

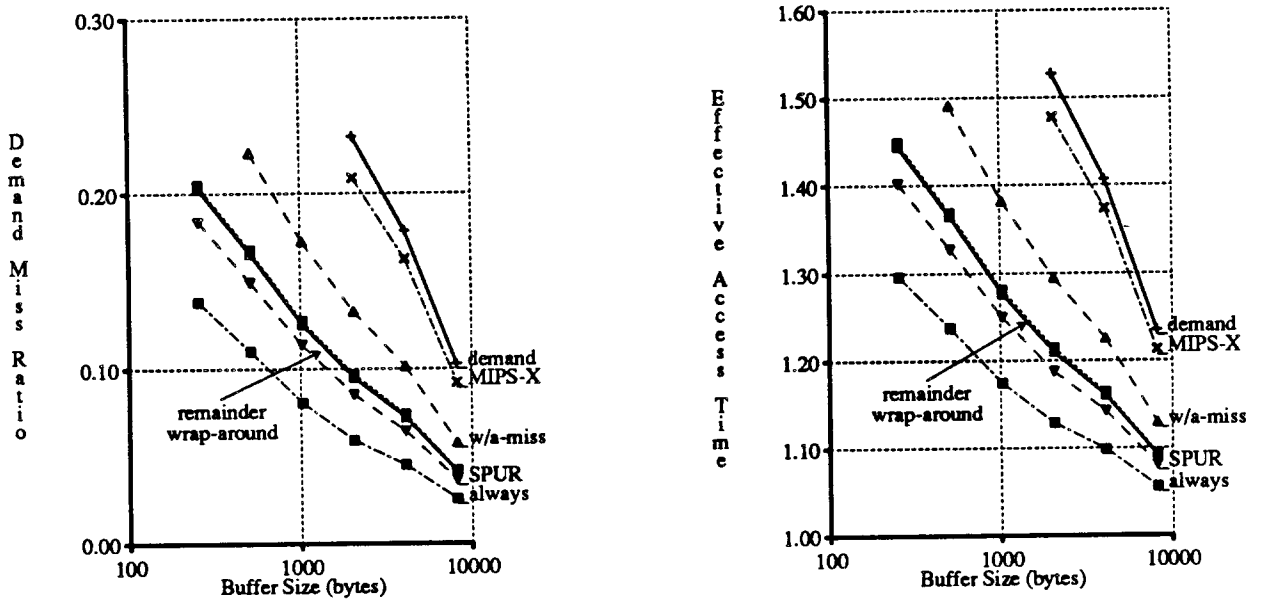


Figure 4-7. Vary Prefetch Algorithm w/ Doubleword Bus.

This figure shows demand miss ratio (left) and effective access time (right) versus IB size for various prefetch algorithms using a single doubleword bus.

As with a singleword bus, SPUR-prefetching with a doubleword bus is preferred for instruction arrays that can be read and written each cycle. Doubling the bus width reduces the effective access time by 12 percent to 1.3270 cycles. The performance of SPUR-prefetch with one doubleword bus is even slightly better (1 to 3 percent) than remainder- and wrap-around-prefetching, because it can get more than one sub-block ahead of the current reference since its prefetch addresses is a function of the address of the last miss, not that of the last reference.

Using two singleword buses rather one singleword bus or using two doubleword buses rather one doubleword bus (see Table 4-4), reduces the effective access times of all prefetch algorithms by varying amounts, but does not change the conclusions drawn above.

4.2.2.3. Reducing IB Miss Penalty

A final approach for improving effective access time is reducing the penalty of IB misses. A miss in the current SPUR IB takes three cycles, two cycles beyond the normal IB access, which are roughly: (1) IB access, (2) external cache access and (3) IB retry. Figure 4-8 shows a phase-by-phase breakdown of an ideal IB miss (repeat of Figure 4-28 in Section 4.6). Three approaches for reducing the miss time, from smallest to greatest implementation cost, are to (1) eliminate the IB and fetch all instructions from the external cache, (2) start the external cache fetch for a non-sequential instruction one cycle early, and

(3) use bypass logic to complete all misses faster.

Cycle/State	Phase	To IB	From IB
1. NORMAL	1	EU sends instrn addr 0x100	
	2		IB detects miss
	3	Cache not busy	IB sends MISS to EU
	4		IB sends 0x100 to Cache
2. MEM_PENDING	1	IB ignores new instrn addr	
	2		IB waiting for Cache
	3	Cache sends instrn(0x100) to MDR	IB sends MISS to EU
	4	IB loads instrn(0x100) from MDR	
3. NORMAL	1	IB ignores new instrn addr	
	2		IB retrys and hits
	3		IB sends instrn(0x100) to EU
	4		

Figure 4-8. Ideal IB Miss.

IB misses cause a delay of at least two cycles. This diagram shows minimum IB miss delay on a reference to address 0x100, which occurs when the external cache is available and can return the instruction word in one cycle. The address is sent to the external cache by the end of cycle 1, phase 4. The instruction word is latched in a memory data register (MDR) near the CPU's data pads at the end of cycle 2, phase 3. The instruction is written into the instruction array in the next phase, and the IB access is retried in cycle 3.

The first approach for reducing miss penalty is to remove the IB and fetch all instructions directly from the external cache. If this fetch can be made in one cycle, then a pipeline results that is similar to RISC II's [Kate83], where an instruction fetch or a data reference is made to the external cache each cycle. Since RISC II was built, however, intra-chip signal speeds have increased more rapidly than inter-chip ones. RISC II accessed external memory in one 330-ns cycle (in 3-micron NMOS); SPUR requires 100 ns for external memory accesses, necessary for loads, stores and IB misses, but only needs 50 ns for an IB access. The external memory access in SPUR completes with an instruction word and a valid bit latched near the address pads. The IB access completes with an instruction word or a MISS[†] instruction latched in the instruction register of the execution unit. If the IB is eliminated, additional time is required on an external memory access to determine if the instruction is valid and move it or a MISS instruction to the execution unit. Thus, it is difficult to fetch instructions directly from an external cache without increasing the cycle time or changing the pipeline (e.g., using a doubled-delayed branch [Chow87]).

The viability of fetching instructions directly from an external cache is a function of how much it perturbs the cycle time and how large an IB it is compared with. The left plot in Figure 4-9 shows effective access times with the SPUR IB and with instructions fetched directly from the external cache leaving the cycle time unchanged, increased by 10 percent or increased by 20 percent. I believe that removing the IB will increase the cycle time by least 20 percent. If the increase is exactly 20 percent, the current IB could be eliminated with no performance loss; however, removing a larger IB results in a large loss. Removing an 8K-byte IB, for example, reduces performance by 33 percent. As levels of integration make larger IBs possible, the no-IB alternative will become increasingly unattractive.

A second approach for reducing the average IB miss penalty is to treat non-sequential instruction fetches differently than sequential ones. On the cycle after an instruction fetch in the current IB, a

[†] The MISS instruction contains an opcode that directs the execution unit to continue waiting for the current instruction fetch.

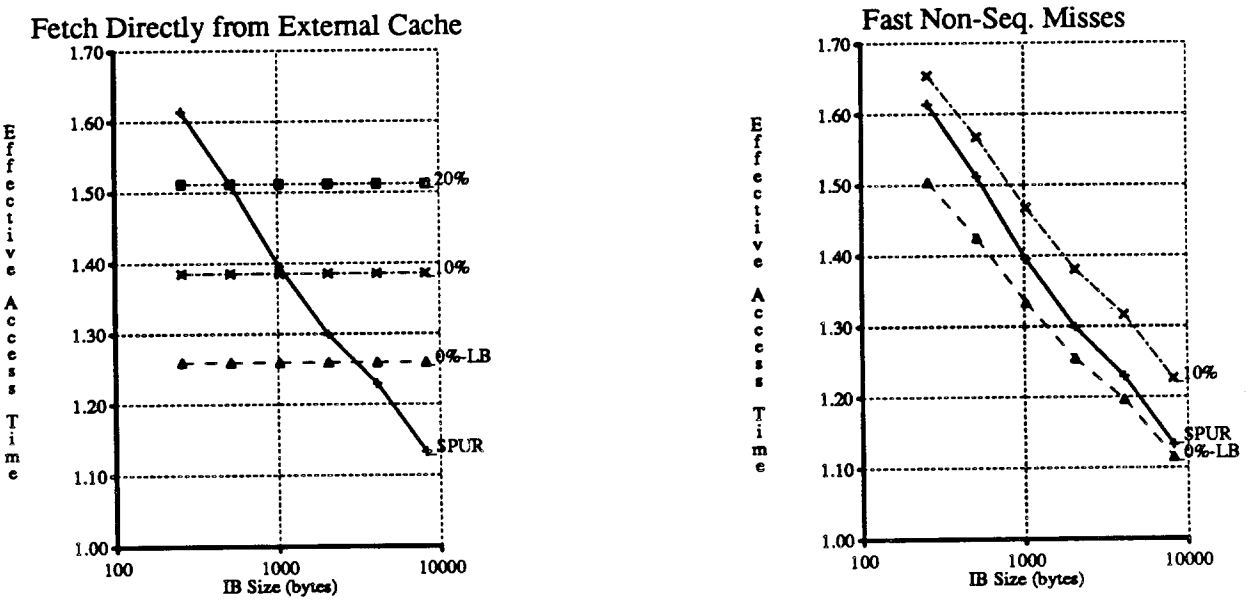


Figure 4-9. Vary Miss Time.

This figure shows effective access times of the SPUR IB and two alternatives that require fewer, longer (by 0, 10 or 20 percent) cycles. Lines whose label includes "LB" are unachievable lower bounds.

The left plot compares the SPUR IB (one-cycle access plus two-cycle miss) with fetching all instructions directly from the external cache in one cycle with no contention. When data references are contending for the external cache, a instruction fetches is delayed until the next idle cache cycle. Clearly, the *no-IB* alternative loses when compared against large IBs.

The right plot compares the SPUR IB with IBs that use fewer cycles for misses on non-sequential references. My estimates of effective access times with this optimization are a bit optimistic, because I do not take into account the effect of prefetches not made when misses are handled one cycle earlier. Nevertheless, results indicated this technique is not worthwhile, since it produces small performance improvements for all but small IBs.

prefetch is done in the block of the last miss, usually the block of the previous reference. If the instruction fetch misses, an external cache access for that instruction begins in the next cycle. For non-sequential instruction fetches, this may not be the best approach since the prefetch in the old block is unlikely to be useful[†], and the non-sequential instruction fetch is more likely to miss than a sequential fetch. Consequently, I propose to reduce the penalty of non-sequential misses from two cycles to one cycle by beginning a non-sequential reference's external cache fetch one cycle earlier, so that the cache fetch is issued concurrently with the IB access rather than after it. One result of moving the fetch up one cycle is that the sequential prefetch that would otherwise have been issued in that cycle cannot be made. The effect of this optimization depends on how often such prefetches would have been useful and on how often the non-sequential fetches miss. If the prefetch would not have been useful, which I expect to be common, then one cycle is saved when the reference misses, and there is no effect if the reference hits. If a useful prefetch is suppressed, then using fast non-sequential misses can cause additional delays of 0, 1 or 2 cycles.

The cost of implementing fast non-sequential misses is three-fold. First, the IB must know when an instruction is probably not sequential. The simplest way to implement this is have the execution unit set a bit when an address is created by the incrementer logic. Sequential instruction addresses produced in other ways (i.e, a jump one word forward or a delayed jump two words forward) will be handled

[†] A prefetch is *useful* if it reduces the demand miss ratio, which means that it loads an instruction not already present, not loaded by a subsequent prefetch, and used before it is replaced. Note that memory pollution [Smit78] is of no concern here, since prefetches do not cause other instructions to be replaced.

correctly, but not with maximum prefetch efficiency. Second, a non-sequential fetch must override the sequential prefetch. This requires a 32-bit multiplexor controlled by the "definitely-sequential" bit. Third, the IB must be able to write the instruction from the non-sequential fetch into the instruction array two phases after it detects a non-sequential miss, since the write moves up from cycle 2, phase 4 to cycle 1, phase 4 and misses are detected in cycle 1, phase 2 (see Figure 4-8). This also implies that the corresponding tag be updated and other instruction words of the block be invalidated more rapidly, at a non-trivial cost.

Unfortunately implementing fast non-sequential misses is not cost-effective, since simulations show the potential performance improvement to be too small to justify the implementation effort. The right plot in Figure 4-9 shows a lower bound on the effective access time achieved with this technique. The bound was calculated by subtracting one cycle for each miss on a non-sequential reference from the total cycles to process a trace. The bound is optimistic, principally because it is calculated without suppressing useful prefetches that are not done during non-sequential misses. Even so using fast non-sequential misses reduces effective access time by less than 6 and 2 percent at 512 and 8K bytes, respectively. The improvement, especially for larger IBs, is too small to justify the implementation effort. I did not construct a more accurate model of using fast non-sequential misses, as the bound clearly shows that its potential contribution to performance is too small to justify its implementation costs.

A third approach for reducing the IB miss penalty is using *fetch bypass* to eliminate the third cycle of a miss, the IB retry, by passing the instruction returned at the end of cycle 2, phase 3 directly to the execution unit. Specifically, this means that at the end of phase 3 the instruction fetched or a MISS instruction must be valid in the execution unit. At the end of phase 3 in the current implementation, only a word-valid bit and an instruction word are valid at the latches adjacent to the CPU data pads. Since the external cache access is on a critical path that determines the CPU's cycle time, using one-cycle misses would lengthen the cycle time. Another drawback of one-cycle misses is that sequential prefetch after a miss would not be in the IB when the IB is accessed for the first instruction after the miss. Consequently bypass logic would be needed to detect that the correct instruction is being prefetched, pass it is on to the execution unit, and suppress the instruction fetch miss.

Using one-cycle misses improves performance of small IBs, but will only improve the performance of larger IBs if it does not increase the cycle time more than a minute amount. Figure IB_MISS_TIME2 shows effective access times with the SPUR IB and with an IB that has the bypass logic for misses and prefetches. I assume this change leaves the cycle time unchanged, increased by 10 percent, or increased by 20 percent. For a 10 percent increase in the cycle time, which I think is reasonable, using one-cycle misses reduces effective access time at 512 bytes by 6.6 percent, but increases it at 8K bytes by 4.4 percent. Even if no cycle time increase occurs, using one-cycle misses results in a 10 percent decrease in effective access time only if the IB's miss ratio is greater than or equal to 0.1250 ($(1-10\%)(1+2m) = 1+1m$). Thus, the effective access time reduction will be smaller than 10 percent for IBs larger than 2K bytes with one singleword bus or larger than 512 bytes with one doubleword bus. Furthermore, implementing this optimization without increasing the cycle time will get more difficult as technological advancements make the implementation of single-cycle external accesses more difficult.

Consequently, none of the three approaches for reducing the number of miss cycles (no-IB, fast non-sequential misses, and one-cycle miss) is worthwhile, especially in the future as IB sizes increase.

4.2.2.4. An Improved SPUR IB

The earlier results of this section point to three cost-effective improvements of the SPUR IB, namely, increasing bus width, IB size and block size. The SPUR IB has an average effective access time of 1.5116 cycles per instruction. Doubling the bus width to 64 bits reduces effective access time by 12.2 percent to 1.3270. Doubling cache size to 1K-bytes further reduces it by 5.8 percent to 1.2499. Doubling block size to 64 bytes (16 instruction words) only reduces effective access time by 2.7 percent to 1.2159, but can reduce the access time by cutting the number of address tags in half (smaller memories can often be accessed more rapidly). The three changes together reduce IB miss ratio by 56

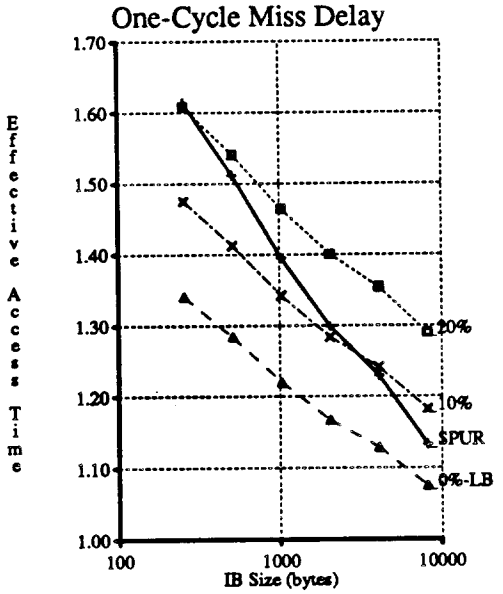


Figure 4-10. Vary More Miss Times.

This figure shows effective access times for SPUR vs. alternates that using one-cycle misses, but with 0, 10 or 20 percent longer cycle times. The line whose label includes "LB" is an unachievable lower bound. A one-cycle miss delay significantly improvements the performance of larger IBs only if reducing miss delay does not increase the cycle time.

percent (0.2276 to 0.0999) and effective access time by 19.6 percent. Using Gibson's assumptions as I did at the end of Section 4.1.3, I find that incorporating these changes in SPUR reduces cycles per instruction by 15.5 percent and increases speed by 23.6 percent. Further increases in IB size can be considered if they do not increase cycle time or reduce yield too much. Each of the next three doublings of IB size reduces effective access time by approximately 4 percent.

Neither of the first two proposed changes, increasing the sub-block or IB size, was possible in the first implementation of SPUR after we decided to save design time by implementing the IB's instruction array with the register file cells. The final change, increasing the block size, is not necessary in SPUR since the tag lookup time does not determine the cycle time.

4.3. Target Instruction Buffers

In this section I examine *target instruction buffers* (TIBs). As with my study of IBs, I first propose a SPUR TIB as a design operating point, and then use trace-driven simulation to study it and similar designs. Results show that using a TIB in SPUR yields an effective access time less than that achieved using the SPUR IB (1.40 vs. 1.51 cycles per instruction). As with the SPUR IB, the most effective way to improve TIB performance is to double the width of the off-chip bus and cache so that instruction prefetching bandwidth is abundant. Doing so reduces effective access time by 12 percent to 1.23 cycles per instruction. Unlike with IBs, however, increasing TIB size is not a cost-effective way to reduce effective access time. No doubling of TIB size improves effective access time by more than 4 percent.

4.3.1. SPUR TIB/PB Architecture and Implementation

To evaluate TIB designs, I propose a TIB and PB for SPUR that has implementation characteristics similar to the SPUR IB. Figure 4-11 displays the architecture of this proposal that I will heretofore refer to as the *SPUR TIB*. Two problems in selecting a TIB/PB similar to the SPUR IB are selecting TIB block size and TIB size. Since a TIB prefetches off the end of a block, while an IB prefetches

within a block, TIBs prefer smaller block sizes. I choose a four-instruction block size for the SPUR TIB, the same as the Am29000 TIB's block size and half of the SPUR IB's block size. The area required for a TIB that holds the same number of instructions as the SPUR IB (128 instructions) is clearly greater than that of the SPUR IB, because the TIB requires twice as many address tags (for its more numerous smaller blocks) and additional area for prefetch buffers and associated logic. A half-sized TIB, on the other hand, uses less area than the SPUR IB, as the prefetch buffer area is smaller than that of half the instruction array. I choose to make the SPUR TIB half-sized so that it could be substituted into the current SPUR CPU layout without other changes. Nevertheless, performance results given here are for TIB sizes 256 though 8K bytes.

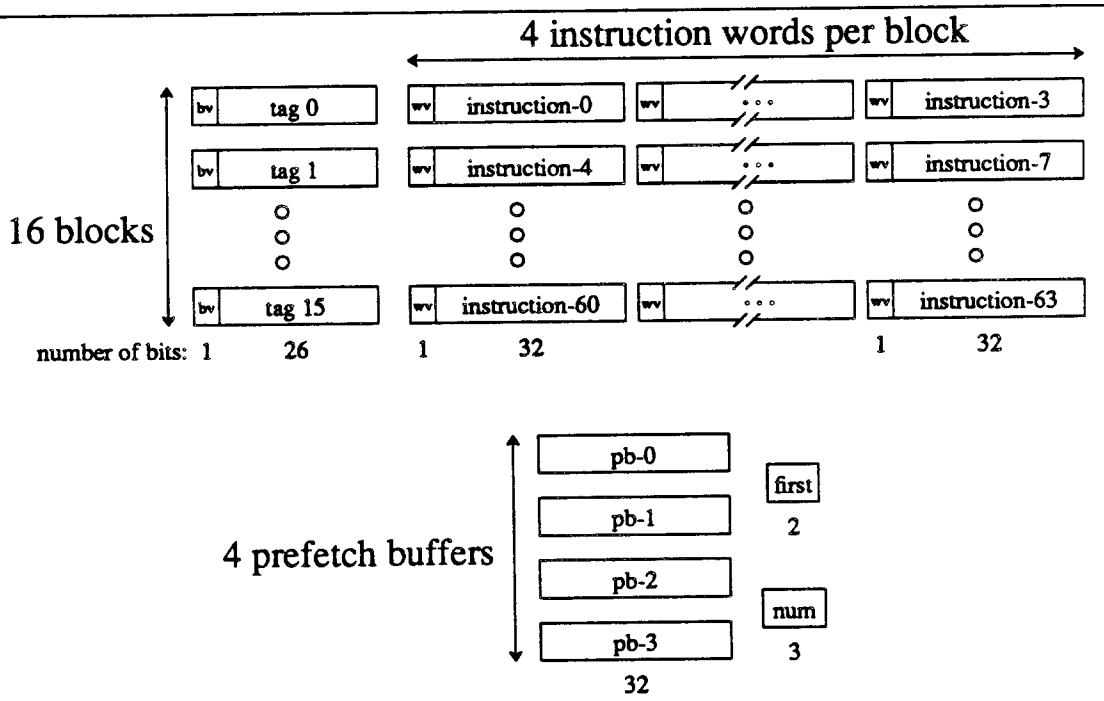


Figure 4-11. SPUR TIB Architecture.

The proposed SPUR target instruction buffer (TIB) is a 256-byte (64-instruction) buffer, divided into 16 direct-mapped blocks. Each block contains a *block-valid* bit (labeled *bv*), a 26-bit address tag, four *word-valid* bits (labeled *wv*) and four 32-bit instructions words. The block-valid bit is asserted to indicate a valid address tag. The address tag holds bits 31-9 and 4-2 of a 32-bit byte address (the LSB is 0 in SPUR). Bits 0 and 1 are discarded because all instruction addresses are on aligned 4-byte boundaries; bits 5 through 8 are used to select one of the 16 blocks. Each word-valid bit indicates whether the corresponding instruction word is valid. The first instruction word in a block holds the instruction at the address stored in the tag. The i-th word holds the instruction at the tag address plus $i-1$ words.

The proposed SPUR prefetch buffer (PB) holds up to four instructions. No tags are needed since I assume the PB is accessed only when the execution unit indicates an address is sequential. The 2-bit register *first* points to the next sequential instruction; 3-bit register *num* indicates the number of valid prefetch buffers (0 to 4). The SPUR TIB holds half the instructions of the SPUR IB so that it could be incorporated into the SPUR CPU layout without other changes. A TIB with the same number of instructions requires more chip area than the SPUR IB to hold twice the number of address tags plus the prefetch buffers.

Figures 4-12 and 4-13 show a possible timing for TIB/PB that corresponds closely to the timing for the SPUR IB (see Figures 4-27 through 4-31 in Section 4.6). Both pay two-cycle penalties on demand misses and can prefetch one instruction per cycle unless blocked by data references. I assume, however, that PB prefetches are bypassed so that an instruction fetch of the same address as an outstanding prefetch will stall the execution unit only until the prefetch returns the instruction. In the SPUR IB pending prefetches are ignored; therefore, the same situation would cause a demand miss and a two-cycle penalty. We do not bypass prefetches in the SPUR IB, because doing so requires a second

32-bit comparator and some additional control complexity. Bypassing prefetches in a prefetch buffer is simpler to implement, because no additional comparator is needed (all instructions are sequential), and the PB can stall the TIB and execution unit on bypassed prefetches with the same mechanism used to stall them on PB and external cache misses.

Cycle/State	Phase	To TIB/PB	From TIB/PB
1. NORMAL	1	EU sends non-sequential instrn addr 0x100	
	2		TIB detects hit & reads instrn(0x100)
	3	*Cache sends instrn(0x84) to PB*	TIB sends instrn(0x100) to EU
	4	TIB may load instrn(0x84) from PB	*PB sends 0x88 to Cache*
2. PB_RESET	1	EU sends sequential addr 0x104	*PB is reset*
	2		instrn(0x104) read from TIB
	3	*Cache sends instrn(0x88), but it is ignored*	TIB sends instrn(0x104) to EU
	4		*PB sends 0x114 to Cache*
3. NORMAL	1	EU sends sequential addr 0x108	
	2		instrn(0x108) read from TIB
	3	*Cache sends instrn(0x114) to PB*	TIB sends instrn(0x108) to EU
	4	TIB loads instrn(0x114) from PB	*PB sends 0x118 to Cache*

Figure 4-12. Ideal TIB Hit.

This figure and Figure 4-13 show TIB timing with the notation used to give IB timing in Figures 4-27 through 4-31 in Section 4.6.

A non-sequential reference that hits in the TIB (0x100 here) causes the PB to be reset, but costs no additional cycles. In cycle 1, the TIB processes the hit, and PB (in *italics*) completes a prefetch and initiates another prefetch for instructions in the previous instruction run (0x84 and 0x88). In cycle 2, the TIB processes the second instruction of the run (0x104), and the PB is reset and issues a prefetch for the instruction after the end of the TIB block of 4 instructions (0x114 = 0x100 + 0x10 + 0x4). Cycle 3 shows the PB loading the instruction at 0x114, which assumes an external cache hit, and issuing a new prefetch to 0x118. After Cycle 4 (not shown), sequential fetches will be handled by the PB since the TIB block is exhausted.

In many ways the SPUR TIB/PB is similar to the Am29000's TIB/PB [Adva87]. Both PBs contain four instruction words managed as a circular buffer, and both TIB have blocks that contain 26-bit address tags and four-instruction blocks with word-valid bits. The principal differences between the two designs is that the Am29000's TIB is larger (32 two-way set-associative blocks vs. SPUR's 16 direct-mapped blocks), and the Am29000 uses separate off-chip buses for instruction and data, whereas SPUR uses a single bus.

4.3.2. TIB/PB Evaluation

Here I examine TIBs similar to the SPUR TIB. I first examine varying TIB size, associativity, index bits, and PB size; then I look at the interaction between off-chip bandwidth and TIB block size; and finally I propose an improved TIB. The miss ratios and effective access times discussed in the section are presented in Tables 4-6 and 4-7. I calculate effective access times from miss ratios using the equations given in Section 4.1.3, assuming a constant cycle time of 1.0.

Cycle/State	Phase	To TIB/PB	From TIB/PB
1. NORMAL	1	EU sends non-sequential instrn addr 0x100	
	2		TIB detects miss on 0x100
	3	*Cache sends instrn(0x84) to PB*	TIB sends MISS to EU
	4	TIB may load instrn(0x84) from PB	TIB sends addr 0x100 to Cache
2. TIB_MISS_1	1	TIB/PB ignores new addr	*PB is reset*
	2		TIB prepared for new entry
	3	*Cache sends instrn(0x100) to PB*	TIB sends MISS to EU
	4	TIB loads instrn(0x100) from PB	*PB sends 0x104 to Cache*
3. TIB_MISS_2	1	TIB/PB ignores new addr	
	2		*PB transfers instrn(0x100) to IR*
	3	*Cache sends instrn(0x104) to PB*	TIB sends instrn(0x100) to EU
	4	TIB loads instrn(0x104) from PB	*PB sends 0x108 to Cache*
4. TIB_LOAD_1	1	EU sends sequential addr 0x104	
	2		*PB transfers instrn(0x104) to IR*
	3	*Cache sends instrn(0x108) to PB*	TIB sends instrn(0x104) to EU
	4	TIB loads instrn(0x108) from PB	*PB sends 0x10c to Cache*

Figure 4-13. Ideal TIB Miss.

A non-sequential reference that misses in the TIB causes the PB to be reset and costs two additional cycles. Cycle 1 shows the TIB detecting the miss and sending the address, 0x100, to the external cache; the PB (in *italics*) completes a prefetch in the previous instruction run (0x84). In cycle 2 the TIB writes the tag and valid bits for a new block, sends MISS to the execution unit, and stores the first instruction of the run into the block; the PB is reset, catches the instruction that missed from the cache and passes it to the TIB, and initiates the next sequential prefetch to 0x104. In cycle 3, the TIB passes instruction that missed to the execution unit and loads the result of the PBs first sequential prefetch; meanwhile, the PB continues prefetching at 0x108. In cycle 4, the TIB accepts a new sequential address and returns the instruction to the execution unit. PB prefetching and TIB loading continues. After the TIB block is full (not shown), execution continues similarly, but without the TIB load.

4.3.2.1. TIB Size, Associativity, Indexing, and PB Size

The TIB feature having the strongest effect on TIB miss ratio, TIB size, does not have the strongest effect on TIB/PB effective access time. The left graph in Figure 4-14 shows the miss ratios for TIBs of various sizes. The SPUR TIB, with sixteen 16-byte blocks (256 bytes), has miss ratio of 0.6457. Doubling TIB size reduces the TIB miss ratio by 18.4 percent to 0.5268; the corresponding effective access time reduction, however, is only 3.5 percent (1.4002 to 1.3515 cycles per instruction). Further doublings of TIB size reduce effective access time by 4.0, 2.6, 2.0 and 2.6 percent. The effective access time reductions are smaller, because all instruction fetches require at least one cycle and, to first-order, reducing the TIB miss ratio does not improve performance on sequential fetches. Symbolically, a ratio of relative effective access time changes to relative miss ratio changes is:

$$\frac{\Delta t_{eff} / t_{eff}}{\Delta m / m} \rightarrow \left[\frac{m}{t_{eff}}\right]\left[\frac{\partial t_{eff}}{\partial m}\right]$$

To first order, the latter partial derivative is the miss penalty in cycles times the fraction of non-sequential references, which for our assumptions and these traces are 2 and 15.5 percent, respectively. Thus:

$$\left[\frac{m}{t_{eff}}\right]\left[\frac{\partial t_{eff}}{\partial m}\right] \approx \left[\frac{m}{t_{eff}}\right] * 2 * 0.155 \ll 0.310.$$

TIB Miss Ratios						
Distinguishing Attribute(s)	Size (bytes)					
	256	512	1024	2048	4096	8192
SPUR TIB	0.6457	0.5268	0.3970	0.3100	0.2521	0.1815
Associativity						
1	0.6457	0.5268	0.3970	0.3100	0.2521	0.1815
2	0.5919	0.4197	0.2996	0.2131	0.1545	0.1204
4	0.5665	0.3760	0.2677	0.1807	0.1253	0.0957
8	0.5618	0.3650	0.2571	0.1716	0.1160	0.0894
LSB of Index						
2	0.6816	0.5490	0.3710	0.2702	0.1873	0.1345
3	0.6655	0.4768	0.3698	0.2645	0.1984	0.1552
4	0.6457	0.5268	0.3970	0.3100	0.2521	0.1815
5	0.7055	0.5955	0.5067	0.4560	0.3874	0.3687
PB Size (bytes)						
∞	0.6457	0.5268	0.3970	0.3100	0.2521	0.1815
32	0.6457	0.5268	0.3970	0.3100	0.2521	0.1815
16	0.6457	0.5268	0.3970	0.3100	0.2521	0.1815
8	0.6457	0.5268	0.3970	0.3100	0.2521	0.1815
Block Size (bytes) w/ One Singleword Bus						
4	0.3970	0.3100	0.2521	0.1815	0.1642	0.1570
8	0.5268	0.3970	0.3100	0.2521	0.1815	0.1642
16	0.6457	0.5268	0.3970	0.3100	0.2521	0.1815
32	0.7901	0.6457	0.5268	0.3970	0.3100	0.2521
One Doubleword Bus						
4	0.3970	0.3100	0.2521	0.1815	0.1642	0.1570
8	0.5268	0.3970	0.3100	0.2521	0.1815	0.1642
16	0.6457	0.5268	0.3970	0.3100	0.2521	0.1815
32	0.7901	0.6457	0.5268	0.3970	0.3100	0.2521
Two Buses						
4	0.3970	0.3100	0.2521	0.1815	0.1642	0.1570
8	0.5268	0.3970	0.3100	0.2521	0.1815	0.1642
16	0.6457	0.5268	0.3970	0.3100	0.2521	0.1815
32	0.7901	0.6457	0.5268	0.3970	0.3100	0.2521

Table 4-6. TIB Miss Ratios.

Parameters not listed match those of the SPUR TIB/PB, described in Section 4.3.1.

Consequently, increasing TIB size yields smaller relative improvement is effective access time than in miss ratio. Given the costs of increasing TIB size, large TIBs, e.g., greater than 1K bytes, are probably not worth it. In addition, doubling TIB size can hurt performance if it causes a cycle time increase greater than a few percent.

Another way to reduce TIB effective access time is to increase TIB associativity. Results in Table 4-7 show that increasing associativity from direct-mapped to 2-way set-associative improves effective access time by 2 to 4 percent, comparable to the improvement achieved by doubling TIB size. Further increases in associativity result in very small improvement. As with IBs, these improvements will be reduced or made negative if they result in even small increases in machine cycle time. Consequently, only the first doubling of associativity is arguably profitable, and it is only so if it does not increase cycle time. Associativity results also indicate that AMD's two-way set-associativity TIB has a 3.2 to 6.9 percent smaller effective access time than a direct-mapped TIB with the same cycle time. Thus AMD's design is reasonable, unless they could have build a faster chip with a direct-mapped TIB.

Whenever caches are not fully-associative, some function must be used for *indexing*, i.e., selecting the set of a reference. With the simplest function for doing this, called *bit selection*, a sub-set of the address bits are used. The most commonly-used bits are the lowest order bits not in the block offset. Bits within the block offset cannot be used, because two references to the same block, but different offsets, must still map to the same set. The lowest order of the remaining bits are used to exploit spatial

TIB Effective Access Times						
Distinguishing Attribute(s)	Size (bytes)					
	256	512	1024	2048	4096	8192
SPUR TIB	1.4002	1.3515	1.2982	1.2640	1.2390	1.2068
Associativity						
1	1.4002	1.3515	1.2982	1.2640	1.2390	1.2068
2	1.3780	1.3099	1.2604	1.2229	1.1981	1.1827
4	1.3704	1.2933	1.2481	1.2104	1.1860	1.1731
8	1.3680	1.2896	1.2437	1.2064	1.1821	1.1703
LSB of Index						
2	1.4158	1.3600	1.2883	1.2445	1.2113	1.1900
3	1.4082	1.3331	1.2875	1.2437	1.2170	1.1979
4	1.4002	1.3515	1.2982	1.2640	1.2390	1.2068
5	1.4179	1.3741	1.3376	1.3152	1.2837	1.2751
PB Size (bytes)						
∞	1.4002	1.3515	1.2982	1.2640	1.2390	1.2068
32	1.4002	1.3515	1.2982	1.2640	1.2390	1.2068
16	1.4002	1.3515	1.2982	1.2640	1.2390	1.2068
8	1.4071	1.3607	1.3096	1.2773	1.2546	1.2262
Block Size (bytes) w/ One Singleword Bus						
4	1.4576	1.4440	1.4348	1.4238	1.4212	1.4201
8	1.4057	1.3651	1.3381	1.3201	1.2982	1.2928
16	1.4002	1.3515	1.2982	1.2640	1.2390	1.2068
32	1.4443	1.3802	1.3233	1.2618	1.2240	1.1945
One Doubleword Bus						
4	1.2527	1.2392	1.2303	1.2196	1.2170	1.2160
8	1.2067	1.1683	1.1423	1.1257	1.1048	1.0998
16	1.2328	1.1955	1.1543	1.1271	1.1086	1.0865
32	1.2764	1.2290	1.1900	1.1454	1.1176	1.0982
Two Buses						
4	1.2155	1.2017	1.1923	1.1812	1.1785	1.1774
8	1.1634	1.1231	1.0961	1.0782	1.0563	1.0509
16	1.2002	1.1634	1.1231	1.0961	1.0782	1.0563
32	1.2450	1.2002	1.1634	1.1231	1.0961	1.0782

Table 4-7. TIB/PB Effective Access Times.

Parameters not listed match those of the SPUR TIB/PB, described in Section 4.3.1.

locality, which states that if block n is in use, other blocks spatially near it, such as blocks $n-1$ and $n+1$, are more likely than otherwise to be in use. If blocks are randomly mapped to S sets, then the probability of two blocks in simultaneous use mapping to the same set is $1/S$. Selecting the set with low-order bits reduces this probability since blocks $n-1$, n and $n+1$ never map to the same set. Selecting a TIB set is different from selecting a cache set, since TIB blocks are not aligned and therefore have no block offset. Consequently, the low-order bits of a word address (bits 2 and above of a byte address with LSB 0) can be used to select the set. It is not clear that selecting with bits 2 and above is optimal, however, since doing so prevents branch targets in adjacent words from colliding, which is unlikely in any case. For the reason I consider selecting a TIB set with higher-order bits. The only other data on this subject that I am aware of is in [Low87]. They find that beginning the index with bit 2 is slightly preferred to beginning it with bit 4, but that all TIB miss ratio differences are less than five percent.

Similarly, our results show that selecting a TIB set with bits beginning at bit 2, 3 or 4 yield comparable performance in direct-mapped TIBs, but beginning with bit 5 produces inferior results, particularly for large TIBs (see Figure 4-15). Since the TIB implementation cost is not affected by which bits select sets, there is no reason not to use the bits that yield the minimum effective access time. I find that 256-byte (16-block) and 512-byte TIBs should be indexed with bit 3 and above, 1K- and 2K-byte TIBs with bit 2 or bit 3 and above, and 4K- and 8K-byte TIBs with bit 2 and above.

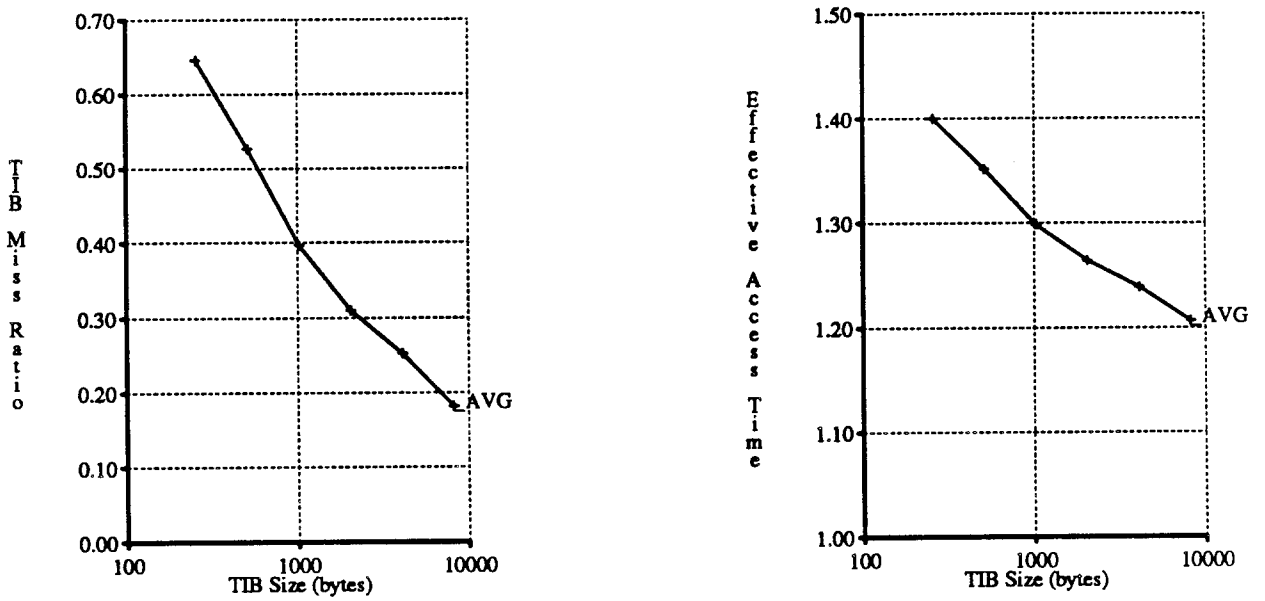


Figure 4-14. Averages for SPUR TIB.

This figure shows demand TIB miss ratio (left) and TIB/PB effective access time (right) versus TIB size. Effective access time results assume a 1-cycle hit, 2-cycle miss penalty and a constant cycle time regardless of TIB size.

Results show large TIBs may not be worth implementing since doubling TIB size only causes a 2 to 4 percent relative improvement in effective access time. Relative improvements in effective access time are smaller than relative improvements in TIB miss ratio since the latter does not include the one-cycle used to access all instructions, or stalls caused by data references interfering with instruction fetches and prefetches.

Most simulations in this section assume an unbounded PB. Results in Table 4-7 show that TIB effective access times are indeed insensitive to PB sizes of 16 bytes and larger, since that size is sufficient to enable prefetching from a single-cycle external cache to stay ahead of a sequential instruction stream. Consequently, ignoring PB size is justified. Grohoski and Patel reached a similar conclusion [Groh82].

4.3.2.2. Off-Chip Bandwidth

As with regular IBs, changes in external bus (and cache) structure greatly affect effective access time with TIBs by altering the available off-chip bandwidth. Figure 4-16 displays results with one singleword bus, one doubleword bus, and two singleword buses. Results with two doubleword buses are not shown since they are exactly the same as those of two singleword buses. Increasing the width of a single bus improves effective access time about 12 percent, independent of cache size by enabling the prefetching to almost always stay ahead of sequential instruction references. With one singleword bus, 0.1999 cycles per instruction are lost by the prefetch buffer associated with a 256-byte TIB waiting for uninitiated prefetches, uncompleted prefetches and data references. With a doubleword bus, these delays are reduced by a factor of 6 to 0.0325 cycles per instruction. Doubling the number of buses further improves effective access time by 2 to 3 percent by reducing the prefetch buffer delays to zero. With two buses, either single or doubleword, the prefetch buffer always stays two instructions ahead of a sequential instruction stream.

Results so far have assumed that all TIB blocks contain 16 bytes (four instructions). Figure 4-17 shows that this design choice is a good one when one singleword bus is used. The effective access time for a TIB with 4-byte blocks is poor, because a one-cycle stall is necessary on all TIB hits. The performance with 8-byte blocks is better since no stalls happen on TIB hits. This TIB block size, however, is too small to let the prefetcher get ahead of the sequential instruction stream; consequently, one cycle is

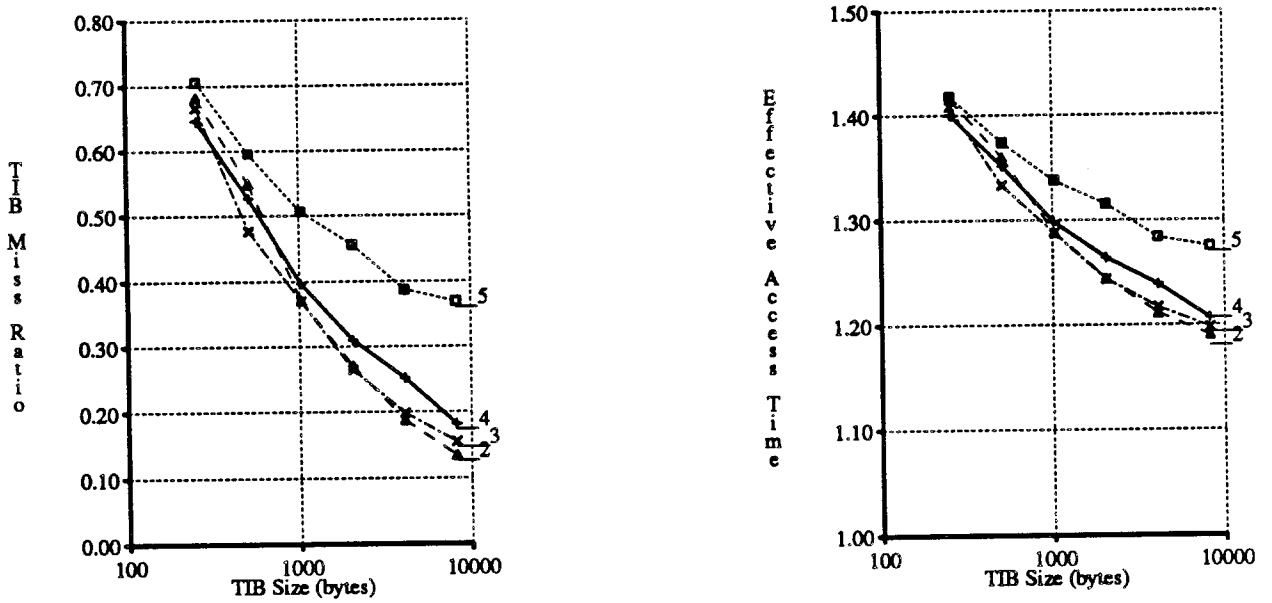


Figure 4-15. Various LSB of TIB Index.

This figure shows demand TIB miss ratio (left) and TIB/PB effective access time (right) versus TIB size for TIB indexed with different bits. Labels show the least-significant bit (LSB) of each index, where 0 is the LSB of a byte-address. Results show indices show begin with bit 3 for TIBs 512 bytes (32 blocks) and smaller, with bit 2 for TIBs 4K bytes and larger, and with either bit 2 or bit 3 in between.

Using one doubleword bus ("1-64") rather than one singleword bus ("1-32") reduces effective access time by around 12 percent. Using two singleword buses ("2-32") rather than one singleword bus reduces effective access time by 2 to 3 percent more than doubling bus width. The effective access time with two doubleword buses (not shown) is exactly the same as that with two singleword buses, because neither option can be stalled by data references.

lost every time a data reference is made. Using 16-byte blocks improves performance over using 8-byte blocks by preventing stalls on the first and second data references after TIB hits. Similarly, the performance for large TIBs with 32-bytes blocks is slightly better since the first six data references after TIB hits do not cause stalls. For small TIBs, however, the performance with 32-byte blocks is worse than that with 16-byte blocks, because the effect of a worse TIB miss ratio induced by the smaller number of larger blocks is more important than the reduced number of data stalls after TIB hits.

Using a doubleword rather than a singleword bus improves the absolute performance for all block sizes (see Figure 4-16) and the relative performance of 8-byte blocks with respect to other block sizes (see Figure 4-18). With one singleword bus, all data references after TIB misses and some data references after TIB hits cause stalls. With one doubleword bus, the performance of all block sizes is improved, because the doubleword bus allows the prefetcher to get ahead of a sequential instruction stream so that few data references, even after TIB misses, stall the pipeline. The performance of 8-byte blocks is improved more than the larger block sizes, because its performance with a singleword bus is particularly sensitive to data references. The performance with 4-byte blocks is still poor, because a stall occurs on every TIB hit.

Using two singleword buses rather than one singleword bus, improves the absolute performance for all block sizes even more than doubling bus width by eliminating all data stalls (see Table 4-7). Using 8-byte blocks yields the smallest effective access time when two buses are used, because 2 instructions are enough on TIB hits, no additional instructions are necessary to cover for data stalls, and the TIB miss ratio with 8-bytes is smaller than that of a TIB with fewer, larger blocks.

The Am29000 has a hybrid external bus structure of separate data and instruction buses that share an address bus. The instruction bus can operate by using the address bus on every reference or just on

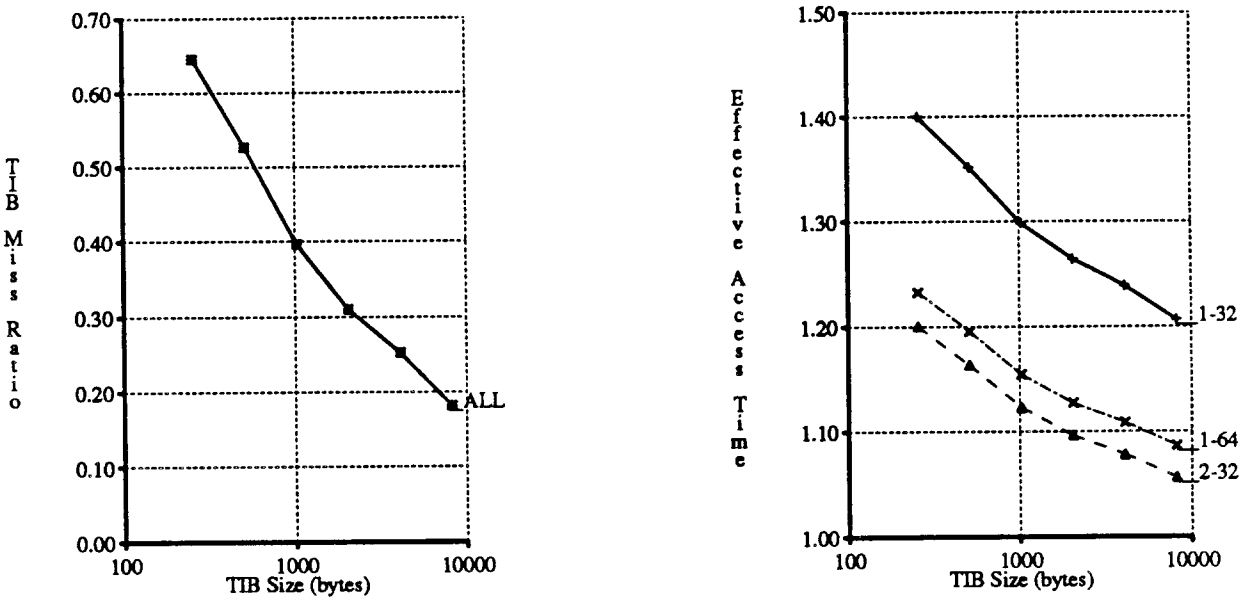


Figure 4-16. Vary External Buses.

This figure shows demand TIB miss ratio (left) and TIB/PB effective access time (right) versus TIB size for various external bus structures. The TIB miss ratio results coincide since they are unaffected by the bus structure.

Using one doubleword bus ("1-64") rather than one singleword bus ("1-32") reduces effective access time by around 12 percent. Using two singleword buses ("2-32") rather than one singleword bus reduces effective access time by 2 to 3 percent more than doubling bus width. The effective access time with two doubleword buses (not shown) is exactly the same as that with two singleword buses, because neither option can be stalled by data references.

every non-sequential reference. In the latter mode, the Am29000 fetches sequential instructions by asserting a bit to ask for the next sequential word and then waiting for the external memory to generate the proper address and return the data. If the address bus is used on every instruction reference, performance is similar to having just one singleword bus; if it used only on non-sequential references, than it is similar to having two singleword buses. The performance of the latter case cannot be worse than that of two singleword buses by more than the number of TIB misses that collide with data references in a one singleword bus system. Since this degradation is never more than 0.0252 cycles per instruction for any of the TIB configurations studied, the performance difference between using AMD's bus structure in this mode and using two singleword buses is negligible.

4.3.2.3. An Improved SPUR TIB

This section's simulations show that the effective access time of the SPUR TIB (256 bytes, 16-byte blocks, direct-mapped, one singleword bus) is 1.40 cycles per instruction, and that, as with the SPUR IB, the most straight-forward way to improve performance is to load instructions and data one doubleword bus or two separate buses. Using one doubleword bus, reduces effective access time by 12 percent., while using two separate buses rather than one reduces effective access time by 14 percent. If separate buses are used, effective access time can be reduced by 3 percent by using 8-byte blocks. After bus width as been doubled, only small improvements are possible from increasing associativity to two-way, increasing TIB size, and reducing TIB block size. It is not clear, however, that these changes are worthwhile since they individually yield about a 2 percent reduction in effective access time only if they can be implemented without increasing the cycle time.

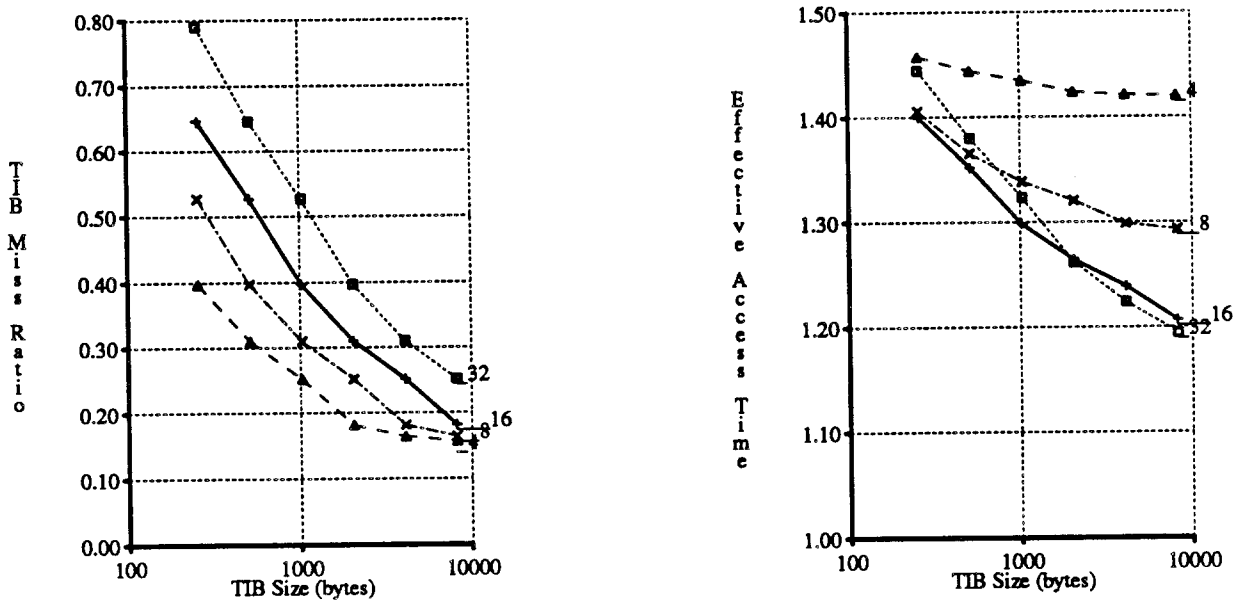


Figure 4-17. Vary Block Size w/ One Singleword Bus.

This figure shows demand TIB miss ratio (left) and TIB/PB effective access time (right) versus TIB size for various TIB block sizes with instructions and data loaded over one singleword bus.

Reducing TIB block size always improves TIB miss ratio, because smaller blocks permit more branch targets to be cached. The smaller blocks do not yield improved effective access times, however, because using smaller blocks makes a TIB more susceptible to prefetch stalls. The effective access time with 4-byte blocks is particularly poor, because one instruction is not enough to cover the latency of restarting sequential prefetching after a TIB hit.

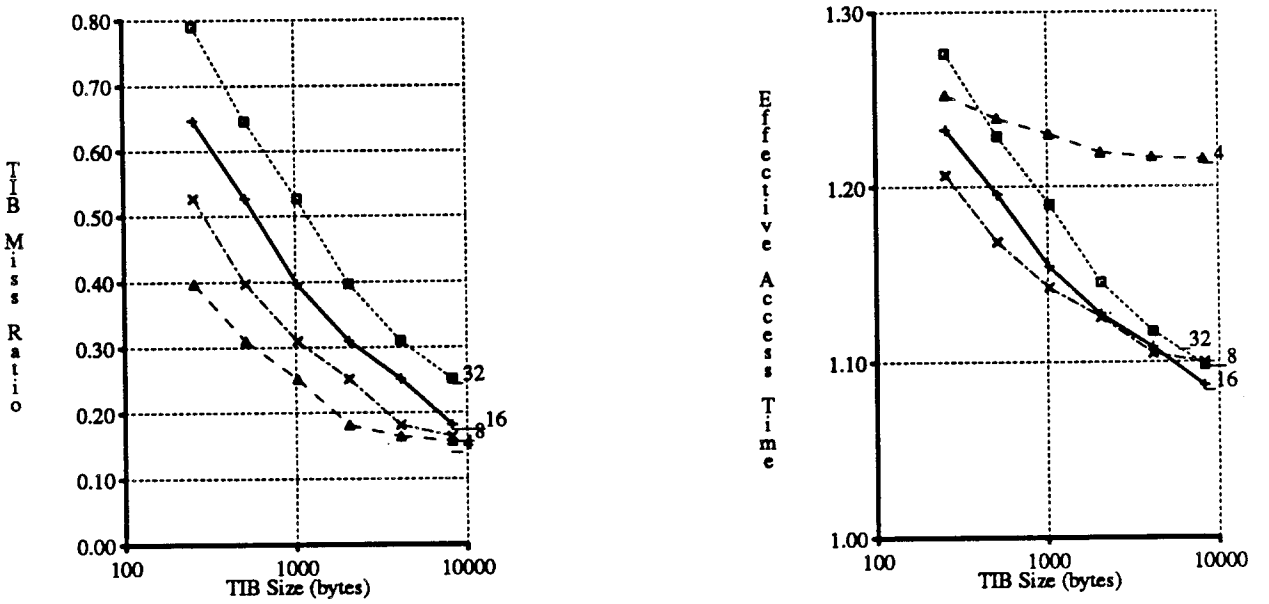


Figure 4-18. Vary Block Size w/ One Doubleword Bus.

This figure shows demand TIB miss ratio (left) and TIB/PB effective access time (right) versus TIB size for various TIB block sizes with instructions and data loaded over one doubleword bus. Results show 8, 16 and 32-byte blocks perform comparatively.

4.4. IBs vs. TIBs

Here I compare the effective access times of IBs and TIBs in systems where external cache references require one or more cycles. A comparison of IB and TIB miss ratios is not useful, because the latter miss ratio ignores sequential references and neither miss ratio takes into account prefetch delays or data interference.

4.4.1. One-Cycle External Cache

The IB and TIB effective access time results, heretofore presented separately, are combined in Figure 4-19. Two key assumptions made are (1) all buffers have the same access (hit) time, and (2) misses cost two cycles and prefetches require one cycle. Assumption (1) tends to make the results displayed in Figure 4-19 favor the buffers that would otherwise tend to have the slower access times, namely, larger buffers with respect to smaller ones, and TIBs with respect to IBs. All things being equal a TIB may have a slower access than an IB of the same size, because it has smaller blocks and therefore more address tags. The tag access may be slower, because larger memories often have slower access times. Assumption (2) implies that all external cache accesses require one cycle. Misses cost two cycles, because I assume a final cycle is necessary inside the CPU chip to complete a miss.

Results with these assumptions show that TIB performance improves with respect to IB performance as buffer size in decreased or as the width or number of buses increases. With a singleword bus, TIB effective access times are worse than that of IBs of buffer sizes ≥ 8K bytes, are comparable for 2K to 4K bytes, and are better at ≤1K bytes. The small TIBs have lower effective access times than the small IBs, because the use of a single-cycle external cache yields a useful prefetch bandwidth of about one instruction per cycle, allowing TIBs to complete prefetches before instructions are needed, while IBs suffer the opportunity cost of caching instructions that can be prefetched without delay. As buffer size increases the performance of IBs improves with respect to TIBs, because, the IB opportunity cost decreases as most useful instructions are cached regardless.

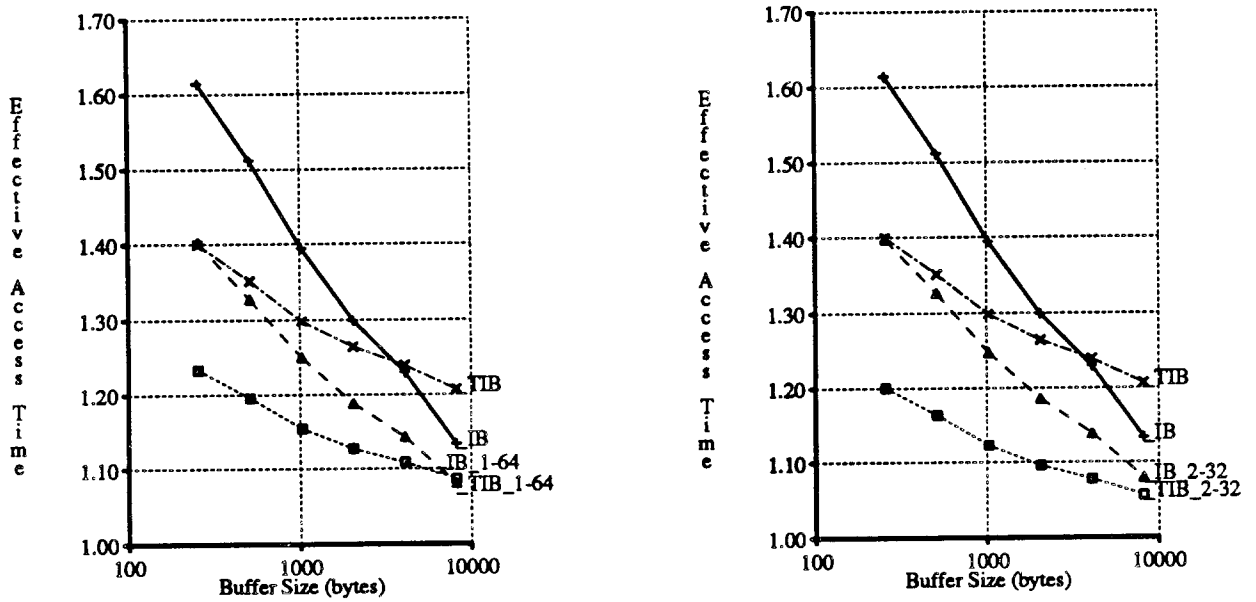


Figure 4-19. IB vs. TIB with Different Buses.

This figure shows effective access time vs. buffer size for instruction buffers ("IB") and target instruction buffers ("TIB"), which interact with external caches via one singleword bus (the default, left and right), one doubleword bus ("1-64", left) and two singleword buses ("2-32", right). Results assume a 1-cycle hit, 2-cycle miss penalty, 1-cycle prefetch, and a constant cycle time regardless of buffer type or size. IBs contain 32-byte blocks whereas TIBs use 16-byte blocks. In practice, the effective access times for larger buffers may be slower than smaller buffers, and TIBs slower than IBs since TIBs have smaller blocks and more tags.

Results with these assumptions show that IBs yield inferior performance for small buffer sizes, say $\leq$ 1K bytes, but yield comparable (within 5 percent) or better performance than TIB for large buffer sizes.

TIB performance improves with respect to IB performance as bus bandwidth increases, because TIBs rely more on prefetching than do IBs. TIBs need more prefetching bandwidth, because, even in the best case, many instructions are never cached. The TIB block used here contains up to four instructions at a branch target (16-bytes); therefore, prefetching is relied on to fetch the fifth and successive instructions of a run before they are needed. The IB block used here contains up to eight total instructions (32-bytes), but on average can hold only 4.5 instructions at and after a branch target at a random alignment (1/8* (1+2+3+...+8)). In the best case, however, subsequent instructions are resident in another IB block.

Another factor to be considered when comparing an IB and TIB of the same number of bytes is that the TIB requires more chip area than the IB to hold its greater number of address tags and its prefetch buffer. A TIB with 16-byte blocks requires twice as many tags as an IB with 32-byte blocks. Furthermore, results in Section 4.2.2.1 imply that IB performance can be improved by doubling the block size to 64 bytes, yielding one quarter the tags of an TIB of the same size. For this reason, the area required for a IB is between that a half-size TIB and one of the same size. I compare IBs with TIB that have different block sizes, because the optimal block size for IBs and TIBs are different. Nevertheless, these comparisons are reasonable, since I use effective access time but not miss ratio.

Table 4-8 shows effective access time change that results from converting an IB to a TIB of half or the same size. Replacing the SPUR IB, for example, results in an performance improvement of between 1.7 and 10.6 percent. As we will see in the next section, however, TIBs are not always superior, particularly when buffer sizes get bigger and external caches get slower.

Relative Effective Access Time Change from IB (row) to TIB (column)		
	1/2 Size TIB	Same Size TIB
SPUR IB	-7.4 %	-10.6 %
SPUR IB w/ doubleword bus	-7.1	-9.9
Improved SPUR IB	-1.7	-5.1

Table 4-8. SPUR IB vs. SPUR TIB.

This table shows the relative performance improvement achieved by replacing each of three SPUR IB designs with comparable TIB designs. The IB designs are: (1) the SPUR IB, which uses a singleword bus and contains 16 blocks of 8 instructions (512 bytes); (2) an IB of the same size, but that uses a doubleword bus; and (3) an IB that contains 16 blocks of 16 instructions and uses a doubleword bus.

These IBs are compared against TIBs that use the same external bus, four-instruction blocks, an unlimited prefetch buffer, and contain half as many or the same number of instructions. Thus, (1) is compared with 256- and 512-byte TIBs that use a singleword bus; (2) is compared with 256- and 512-byte TIBs that use a doubleword bus; and (3) is compared with 512- and 1K-byte TIBs that use a doubleword bus. The area required to implement each pair of TIBs brackets the area needed for the corresponding IB, since the TIBs use more blocks and a prefetch buffer.

Results show that TIBs are preferred to IBs for the buffer sizes relevant to SPUR and assuming one-cycle external cache access; however, the improvement of switching from an IB to TIB decreases as IB performance is improved.

4.4.2. Multiple-Cycle External Cache

The results so far assume a single-cycle external cache. However, semiconductor trends promise to make single-cycle external memories more difficult and more expensive, principally because intra-chip delays are getting smaller more rapidly than inter-chip delays. Consequently, it is of interest to examine IBs and TIBs where instruction misses cost multiple cycles off-chip plus one cycle on-chip, and prefetches cost multiple cycles off-chip plus no cycles on-chip. All effective access times discussed in this section are presented in Tables 4-9 and 4-10.

Two approximations are made in this analysis. First, IBs use remainder-prefetch rather than SPUR-prefetch, because SPUR-prefetch was designed specifically for one-cycle prefetching where a useless prefetch could be issued at no cost. Figure 4-20 recalls results of Section 4.2.2.2, where I show that the performance difference between remainder-prefetch and SPUR-prefetch with a single-cycle external cache is negligible. Second, interference between instructions and data accesses to a multiple-cycle external cache is optimistically modeled as if the external cache is pipelined so that it can accept a new reference each cycle (see Table 4-2 in Section 4.1.3). Consequently results should not be extrapolated to systems with external caches slower than two or three cycles.

Selected results for two- and three-cycle external caches, where my interference model is reasonable, are shown in Figures 4-21 (2 cycles for the first word and 1 cycle for subsequent words), 4-22 (2, 2), 4-23 (3, 1), and 4-24 (3, 3).

Two conclusions follow from these results. First, increasing the delay for the first instruction increases the effective access time for all configurations, but does not significantly change the relative positions of alternatives. A less optimistic modeling of delays induced by data interference could, however, hurt the relative performance of TIBs with respect to IBs, because TIBs require more of the available prefetch bandwidth.

Second, increasing the delay for subsequent instruction prefetches improves IB performance with respect to that of TIBs. Given an external cache that requires two cycles to access the first word and is connected to the CPU with one singleword bus, increasing external cache access time for subsequent words (prefetches) from 1 to 2 cycles improves IB effective access time with respect to that a TIB of the same size by 8 percent at 512 bytes and 11 percent at 4K bytes.

IB Effective Access Times						
Distinguishing Attribute(s)	Size (bytes)					
	256	512	1024	2048	4096	8192
One Bus: first, next.						
1, 1	1.6059	1.5059	1.3921	1.3009	1.2322	1.1349
One Bus: first, next.						
2, 1	1.8749	1.7304	1.5652	1.4336	1.3342	1.1937
2, 2	2.5109	2.2599	1.9770	1.7529	1.5767	1.3337
One Bus: first, next.						
3, 1	2.1439	1.9548	1.7383	1.5662	1.4361	1.2524
3, 2	2.7872	2.4912	2.1549	1.8894	1.6811	1.3942
3, 3	3.4094	3.0073	2.5573	2.2011	1.9189	1.5309
One 64b Bus: first, next.						
1, 1	1.4490	1.3684	1.2814	1.2147	1.1641	1.0939
One 64b Bus: first, next.						
2, 1	1.6537	1.5362	1.4091	1.3118	1.2381	1.1358
2, 2	1.7380	1.6028	1.4535	1.3430	1.2600	1.1504
One 64b Bus: first, next.						
3, 1	1.8583	1.7040	1.5367	1.4090	1.3120	1.1778
3, 2	1.9426	1.7706	1.5812	1.4402	1.3340	1.1923
3, 3	2.2527	2.0265	1.7739	1.5862	1.4429	1.2562
Two Buses: first, next.						
1, 1	1.3853	1.3151	1.2385	1.1817	1.1372	1.0778
Two Buses: first, next.						
2, 1	1.5780	1.4727	1.3577	1.2726	1.2058	1.1168
2, 2	2.2968	2.0736	1.8265	1.6358	1.4832	1.2777
Two Buses: first, next.						
3, 1	1.7706	1.6302	1.4770	1.3635	1.2745	1.1557
3, 2	2.4972	2.2386	1.9511	1.7310	1.5547	1.3186
3, 3	3.2023	2.8255	2.4104	2.0869	1.8272	1.4762

Table 4-9. IB Times w/ Multiple-Cycle External Caches.

This table shows effective access time for IBs similar to the SPUR IB, described in the Section 4.2.1, but with varying buffer size, bus structure and off-chip access times. The numbers in the first column give various off-chips delays, in cycles, for fetching a word (labeled "first") and for fetching subsequent sequential words (labeled "next").

TIB Effective Access Times						
Distinguishing Attribute(s)	Size (bytes)					
	256	512	1024	2048	4096	8192
One Bus: first, next.						
1, 1	1.4002	1.3515	1.2982	1.2640	1.2390	1.2068
One Bus: first, next.						
2, 1	1.5423	1.4733	1.3979	1.3490	1.3136	1.2692
2, 2	2.1362	2.0200	1.8933	1.8081	1.7492	1.6711
One Bus: first, next.						
3, 1	1.6843	1.5951	1.4976	1.4339	1.3883	1.3316
3, 2	2.2382	2.1044	1.9583	1.8599	1.7925	1.7040
3, 3	2.9152	2.7325	2.5320	2.3986	2.3068	2.1899
One 64b Bus: first, next.						
1, 1	1.2328	1.1955	1.1543	1.1271	1.1086	1.0865
One 64b Bus: first, next.						
2, 1	1.3426	1.2867	1.2252	1.1844	1.1568	1.1236
2, 2	1.5213	1.4448	1.3631	1.3102	1.2727	1.2335
One 64b Bus: first, next.						
3, 1	1.4524	1.3778	1.2961	1.2417	1.2050	1.1608
3, 2	1.6373	1.5420	1.4397	1.3728	1.3262	1.2757
3, 3	1.9118	1.8007	1.6872	1.6060	1.5511	1.4840
Two Buses: first, next.						
1, 1	1.2002	1.1634	1.1231	1.0961	1.0782	1.0563
Two Buses: first, next.						
2, 1	1.3004	1.2451	1.1847	1.1442	1.1173	1.0844
2, 2	1.9674	1.8548	1.7340	1.6516	1.5952	1.5185
Two Buses: first, next.						
3, 1	1.4005	1.3267	1.2463	1.1923	1.1563	1.1126
3, 2	2.0676	1.9365	1.7956	1.6996	1.6343	1.5467
3, 3	2.8943	2.7040	2.4981	2.3631	2.2696	2.1485

Table 4-10. TIB Times w/ Multiple-Cycle External Caches.

This table shows effective access time for TIBs similar to the SPUR TIB, described in the Section 4.3.1, but with varying buffer size, bus structure and off-chip access times. The numbers in the first column give various off-chips delays, in cycles, for fetching a word (labeled "first") and for fetching subsequent sequential words (labeled "next").

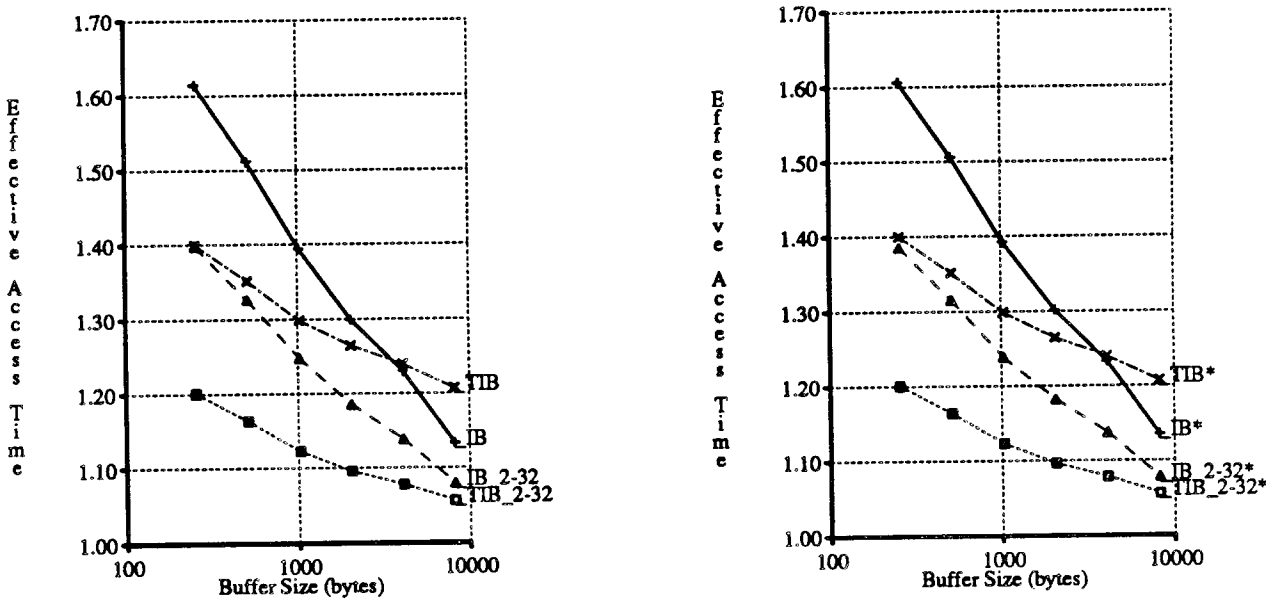


Figure 4-20. Actual vs. Approximate.

This figure shows effective access time vs. buffer size for instruction buffers ("IB") and target instruction buffers ("TIB"), which interact with external cache(s) via one singleword bus (the default) and two singleword buses ("2-32"). Results assume a 1-cycle hit, 2-cycle miss penalty (1 cycle external plus 1 cycle on-chip), 1-cycle prefetch (1 cycle external plus no cycles on-chip) and a constant cycle time regardless of buffer type or size.

The left plot shows actual IB and TIB results, while the right plot displays approximate results ("*") whose IBs use remainder-prefetch rather than SPUR-prefetch. The differences between actual and approximate results are negligible.

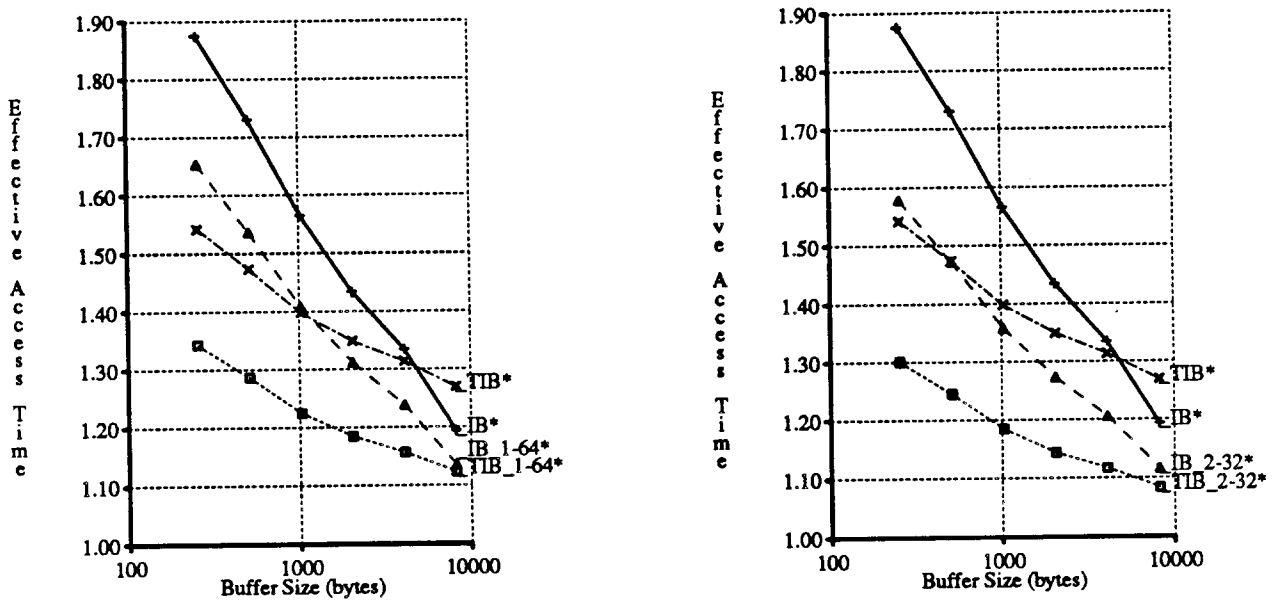


Figure 4-21. 2 Cycles for First Word; 1 for Subsequent Words.

This figure shows effective access time vs. buffer size for instruction buffers ("IB") and target instruction buffers ("TIB"), that interact with external cache(s) via one singleword bus (the default, left and right), one doubleword bus ("1-64", left), and two singleword buses ("2-32", right). Results assume an external cache requires two cycles for the *first* word fetched and one cycle for *subsequent* words. Results with one bus are slightly optimistic since some interference is not modeled.

Results show that increasing the delay for the first word, but not subsequent words, increases all effective access times, but does not change the relative positions of alternative designs.

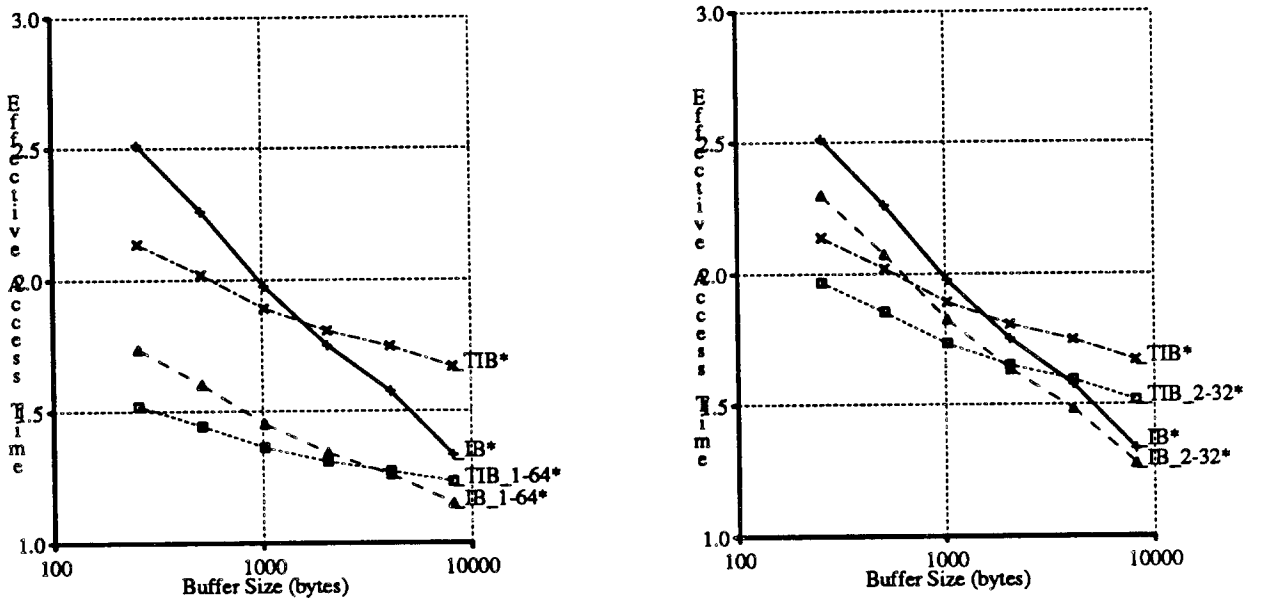


Figure 4-22. 2 Cycles for First and Subsequent Words.

This figure shows effective access time vs. buffer size for instruction buffers and target instruction buffers interacting with an external cache that requires two cycles for the first word fetched and two cycles for subsequent words.

Results show that increasing the delay for subsequent words increases the effective access times of TIBs with respect to IBs, making IBs attractive at smaller sizes.

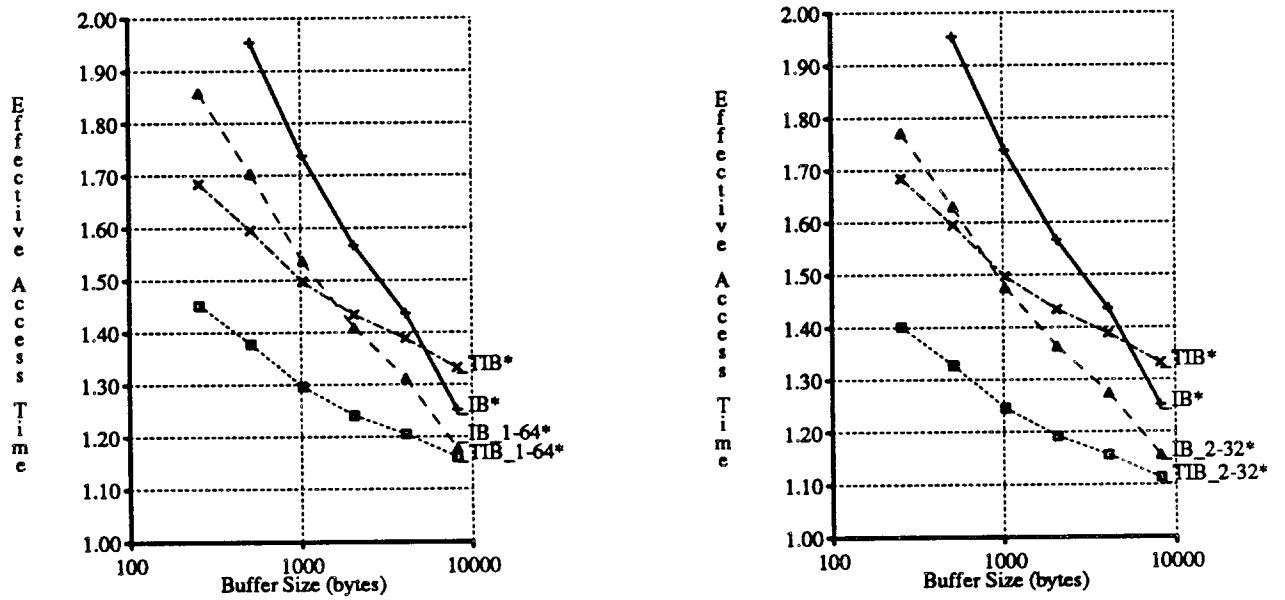


Figure 4-23. 3 Cycles for First Word; 1 for Subsequent Words.

This figure shows effective access time vs. buffer size for instruction buffers and target instruction buffers, that interact with external cache(s) that require three cycles for the *first* word fetched and one cycle for *subsequent* words.

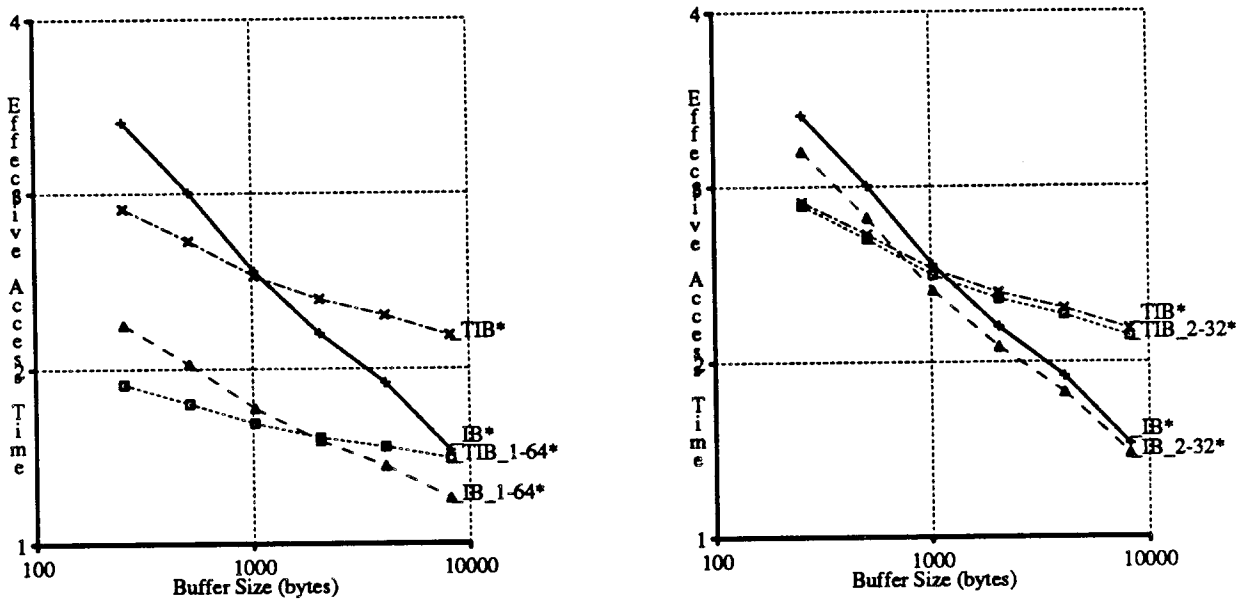


Figure 4-24. 3 Cycles for First and Subsequent Words.

This figure shows effective access time vs. buffer size for instruction buffers and target instruction buffers interacting with an external cache that requires three cycles for the first word fetched and three cycles for subsequent words.

Results show that IBs are preferred to TIBs with a three-cycle external cache, except for buffer sizes less than or equal to 512 bytes.

4.5. Conclusions

I have examined instruction memories, organized as IBs and TIBs, on single-chip RISC microprocessors in systems where these chips are connected to a low-latency external cache (or memory). I find:

(1) The effective access time for IBs is affected most by off-chip bandwidth and buffer size; whereas TIB performance is sensitive to off-chip bandwidth, but less sensitive to buffer size.

(2) TIBs are preferred or comparable to IBs in systems with one-cycle external caches unless buffer size is large (≥ 8K bytes). Consequently, the TIB on the Am29000 microprocessor is a cost-effective use of on-chip memory, and the performance of SPUR can be improved by replacing the SPUR IB with a TIB of comparable size.

(3) IBs are preferred or comparable to TIBs in systems with multiple-cycle external caches where prefetches to take more than one cycle, unless buffer size is small (≤ 512 bytes). Consequently, IBs will be preferred to TIBs in the next generation of CMOS microprocessors, because technological trends are making larger on-chip memories possible, and making it more difficult to build a single-cycle external memory.

These results do not directly extend to systems where RISC CPUs are attached to slow external caches or memories, because (1) the prefetching of sizes larger than one- or two-instruction sub-blocks may be preferred, and (2) my model of instruction and data interference will break down unless external memories are pipelined (at non-trivial expense) to accept a new reference every cycle.

These results do not directly extend to systems with CISC CPUs, since those systems need not fetch an instruction per cycle. Prefetching is such systems is easier to implement, for example, since the results of a prefetch are rarely needed in the next cycle and can be written into the instruction memory during the many idle cycles between instruction fetches.

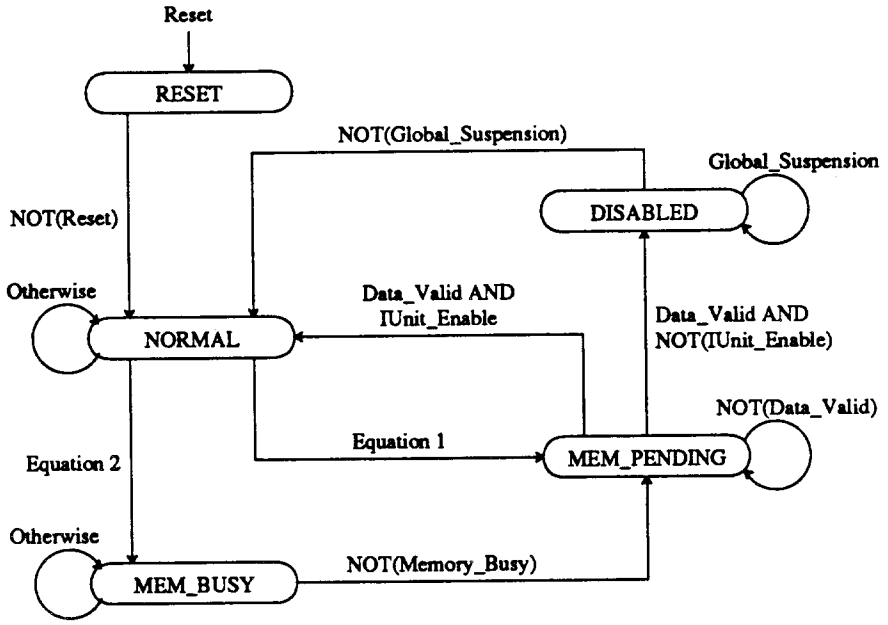
4.6. Appendix: SPUR Instruction Buffer Implementation

The implementation of the SPUR IB began with a register-transfer-level implementation specification done by me in late 1984. This specification was executable, and matched the proposed phase-by-phase timing of the SPUR CPU. A trial VLSI implementation was started in the Spring 1985 by students in a graduate class taught at Berkeley by Prof. Randy Katz [Katz85]. This implementation was discarded, however, because of a change of technology and personnel.

The IB implementation incorporated in the SPUR CPU was done by Rich Duncombe during the 1985-86 academic year with some assistance from Shing Kong, Dave Lee, and myself [Dunc86]. Dave Lee also re-implemented IB control circuitry and been responsible for fixing IB implementation bugs that arose as the IB was integrated with the execution unit and as the CPU was integrated with the external cache. The SPUR IB contains 39,400 transistors and occupies 4200 microns by 6000 microns in a 1.8-micron CMOS technology. It has an access time of 50 ns and a cycle time of 100 ns. The IB memory, shown earlier in Figure 4-3, is implemented in a tag array and an instruction array. The tag array, which holds block-valid bits and address tags, contains 16 entries of 24 bits each. The instruction array, which holds word-valid bits and instructions, is 128 entries of 33 bits each. The IB is controlled by two finite state machines (shown in Figures 4-25 and 4-26), which supervise fetching and prefetching, respectively.

Figures 4-27 to 4-29 show the phase-by-phase operation of the SPUR IB implementation when caching is enabled, but prefetching is disabled. The figures diagram a hit, ideal miss, and slow miss. An *ideal* miss is an IB miss serviced in the minimum two cycles, and therefore not delayed by a data reference or external cache miss.

Figures 4-30 and 4-31 show IB operation on a hit and an ideal miss with prefetching enabled. The prefetch address is reset after every IB miss to be the next word in the block. If the cache is not busy, subsequent words in the block are prefetched before they are needed by sequential instruction execution. Prefetching is not stopped when the end of the block is encountered or when the block is full. Rather we implemented "perpetual" prefetching, because it requires less logic. Redundant prefetches need not be stopped, because prefetches are "free" since they never block other cache accesses.

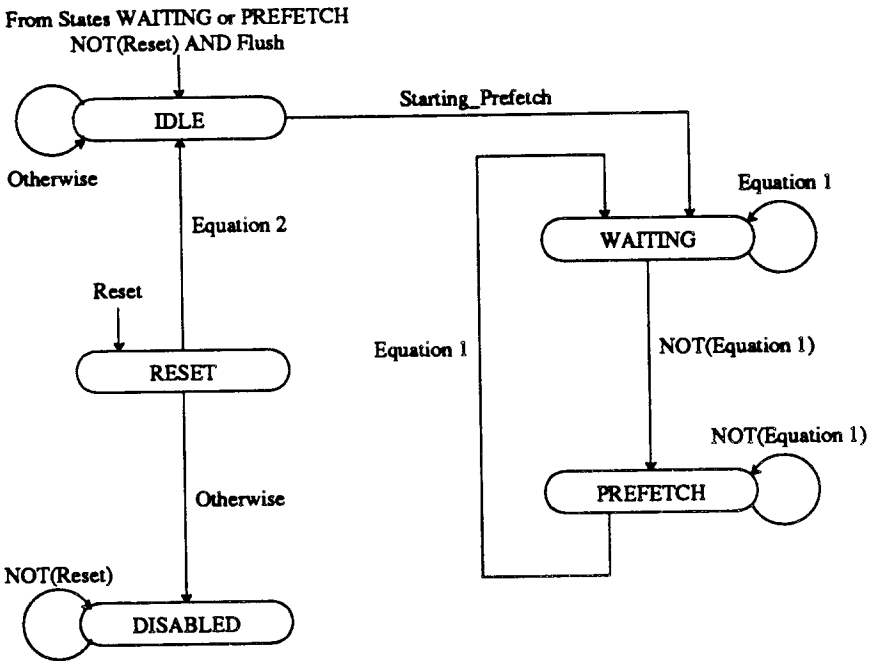


Equation 1: NOT(Global_Suspension) AND (Miss OR Flush) AND NOT(Memory_Busy)

Equation 2: NOT(Global_Suspension) AND (Miss OR Flush) AND Memory_Busy

Figure 4-25. IB Fetch State Diagram.

This diagram, based on Figure 3-1 in [Dunc86], shows the five-state FSM that controls instruction fetch processing. The IB state changes to RESET if IB-reset is asserted, regardless of the previous state. Instruction addresses are accepted from the execution unit only in state NORMAL. The IB stays in state NORMAL as long as hits occur. When a miss occurs, the IB tries to initiate an instruction fetch to the external cache and moves to the state MEM_PENDING. If the external cache is busy (e.g., handling a execution unit data reference), the IB waits in state MEM_BUSY until the fetch can be issued. The IB leaves state MEM_PENDING when the instruction is returned by the external cache. If the IB is enabled, the miss is completed by re-trying the IB access, which hits. If the IB is disabled, state DISABLED is used to transfer the instruction to the execution unit. In either case, the minimum delay for a miss is two cycles beyond the one cycle allotted in the pipeline for instruction fetching.



Equation 1: Starting_Prefetch OR Memory_Busy

Equation 2: NOT(Reset) AND IUnit_Enable AND Prefetch_Enable

Figure 4-26. IB Prefetch State Diagram.

This diagram, based on Figure 3-2 in [Dunc86], shows the five-state FSM that controls instruction prefetching. The state changes to RESET if IB-reset is asserted, regardless of the previous state. If prefetching is disabled, the state changes to DISABLED and the prefetcher does nothing; otherwise, the prefetcher moves to state IDLE and waits for the first IB miss to occur. When it does, the prefetcher moves to state WAITING and prepares to prefetch subsequent words in the block of the IB miss. The prefetcher stays in state WAITING or moves to it whenever the external cache is busy (e.g., processing data fetches) or an instruction fetch miss occurs. The prefetcher stays in state PREFETCH or moves to it whenever the external cache is available. The prefetcher does not leave these two states unless the IB is reset or flushed.

Cycle/State	Phase	To IB	From IB
1. NORMAL	1	EU sends instrn addr 0x100	
	2		IB detects hit
	3		IB sends instrn(0x100) to EU
	4		

Figure 4-27. IB Hit.

This figure diagrams the phase-by-phase activity of the SPUR IB implementation on an IB hit to address 0x100. An IB hit takes two clock phases or half of a cycle. In phase 1, the IB precharges bit lines in the instruction array and latches the instruction address from the execution unit. In phase 2 the instruction and tag arrays are read, and the hit/miss is determined. In phase 3, the IB sends an instruction or MISS to the execution unit. "MISS" is a 32-bit instruction that causes the execution unit pipeline to continue requesting the same instruction.

This diagram assumes that prefetching is disabled. When it is enabled, phase 3 is used to precharge instruction array bit lines, and phase 4 is used to write prefetched instructions into the instruction array. The same timing -- precharge, read, precharge, write -- is used in SPUR's register file.

Cycle/State	Phase	To IB	From IB
1. NORMAL	1	EU sends instrn addr 0x100	
	2		IB detects miss
	3	Cache not busy	IB sends MISS to EU
	4		IB sends 0x100 to Cache
2. MEM_PENDING	1	IB ignores new instrn addr	
	2		IB waiting for Cache
	3	Cache sends instrn(0x100) to MDR	IB sends MISS to EU
	4	IB loads instrn(0x100) from MDR	
3. NORMAL	1	IB ignores new instrn addr	
	2		IB retrys and hits
	3		IB sends instrn(0x100) to EU
	4		

Figure 4-28. Ideal IB Miss.

IB misses cause a delay of at least two cycles. This diagram shows the minimum IB miss delay, which occurs when the external cache is available and can return the instruction word in one cycle. The instruction address (0x100) is sent to the external cache by the end of cycle 1, phase 4. The instruction word is latched in a memory data register (MDR) near the CPU's data pads at the end of cycle 2, phase 3. The instruction is written into the instruction array in the next phase, and the IB access is retried in cycle 3.

Cycle/State	Phase	To IB	From IB
1. NORMAL	1	EU sends instrn addr 0x100	
	2		IB detects miss
	3	Cache busy	IB sends MISS to EU
	4		IB cannot send addr to Cache
2. MEM_BUSY	1	IB ignores new instrn addr	
	2		IB waiting for Cache
	3	Cache busy	IB sends MISS to EU
	4		IB cannot send addr to Cache
3. MEM_BUSY	1	IB ignores new instrn addr	
	2		IB waiting for Cache
	3	Cache not busy	IB sends MISS to EU
	4		IB sends 0x100 to Cache
4. MEM_PENDING	1	IB ignores new instrn addr	
	2		IB waiting for Cache
	3	Cache does not send instrn	IB sends MISS to EU
	4		
5. MEM_PENDING	1	IB ignores new instrn addr	
	2		IB waiting for Cache
	3	Cache sends instrn(0x100) to MDR	IB sends MISS to EU
	4	IB loads instrn(0x100) from MDR	
6. NORMAL	1	IB ignores new instrn addr	
	2		IB retrys and hits
	3		IB sends instrn(0x100) to EU
	4		

Figure 4-29. A Slow IB Miss.

An IB miss can cause a delay of greater than two cycles. One or more MEM_BUSY cycles are added to the time of an IB miss whenever the external cache is not immediately ready to accept an instruction fetch, e.g., because it is handling a data fetch. One or more MEM_PENDING cycles can be added to the minimum one whenever the external cache takes longer than one cycle to handle an instruction fetch, e.g., because of a cache miss. Cycles 2 and 4 may be repeated an arbitrary number of times.

Cycle/State	Phase	To IB	From IB
1. NORMAL	1	EU sends instrn addr 0x100	
	2		IB detects hit
Prefetch= PREFETCH	3	*Cache sends instrn(0x84) to MDR*	IB sends instrn(0x100) to EU
	4	*Prefetcher loads instrn(0x84) from MDR*	*Prefetcher sends 0x88 to Cache*

Figure 4-30. IB Hit with Prefetching.

This figure diagrams the phase-by-phase activity of the SPUR IB implementation for an IB hit with prefetching enabled. The IB access (to address 0x100) and hit proceed as they did in Figure 4-26. Prefetch operations, shown in *italics*, proceed in parallel with the IB fetch. In phase 4, the prefetcher sends a prefetch address to the external cache if that cache is not busy. The prefetch address, 0x84 here, is a function of the address of the last miss, and may be unrelated to the address of the current reference, 0x100. At the end of the next phase 3, the prefetcher latches a word-valid bit and an instruction word. Both are written into the instruction array in the next phase 4. If the external cache cannot provide the prefetched instruction, probably due to a cache miss, it merely returns a word-valid bit set to invalid.

Cycle/State	Phase	To IB	From IB
1. NORMAL	1	EU sends instrn addr 0x100	
	2		IB detects miss
Prefetch= Prefetch	3	*Cache sends instrn(0x84) to MDR*	IB sends MISS to EU
	4	*Prefetcher loads instrn(0x84) from MDR*	IB sends 0x100 to Cache
2. MEM_PENDING	1	IB ignores new instrn addr	
	2		IB waiting for Cache
Prefetch= IDLE	3	Cache sends instrn(0x100) to MDR	IB sends MISS to EU
	4	IB loads instrn(0x100) from MDR	*Prefetcher sends 0x104 to Cache*
3. NORMAL	1	IB ignores new instrn addr	
	2		IB retrys and hits
Prefetch= PREFETCH	3	*Cache sends instrn(0x104) to MDR*	IB sends instrn(0x100) to EU
	4	*Prefetcher loads instrn(0x104) from MDR*	*Prefetcher sends 0x108 to Cache*

Figure 4-31. Ideal IB Miss with Prefetching.

This diagram shows the minimum IB miss with prefetching activity in *italics*. An IB miss to address 0x100 changes the prefetch address to 0x104, the next word within the 32-byte block of 0x100. Prefetching proceeds in the same block until the next IB miss. The prefetch address wrap-arounds to the beginning of the block when the end of a block is reached. More precisely, the prefetch address after an IB miss or prefetch to address x is block(x) + [(x+4) mod 32], where block(x) is $x - (x \bmod 32)$.

4.7. References

[Adva87] Advanced Micro Devices, Am29000 User's Manual (1987).

[Agar87] A. Agarwal, P. Chow, M. Horowitz, J. Acken, A. Salz and J. Hennessy, On-chip Instruction Caches for High Performance Processors, *Proc. Conf. on Advanced Research in VLSI*, Stanford (March 1987).

[Alpe87] D. Alpert, The National NS32532, U.C. Berkeley Systems Seminar (July, 1987).

[Chow87] P. Chow and M. Horowitz, Architectural Tradeoffs in the Design of MIPS-X, *Proc. 14th International Symposium on Computer Architecture*, Pittsburgh (June 1987).

[Digi80] Digital Equipment Corp., VAX Hardware Handbook (1980).

[Ditz87] D. R. Ditzel, H. R. McLellan and A. D. Berenbaum, The Hardware Architecture of the CRISP Microprocessor, *Proc. Fourteenth International Symposium on Computer Architecture*, Pittsburgh (June 1987).

[Dunc86] R. R. Duncombe, The SPUR Instruction Unit: An On-Chip Instruction Cache Memory for a High Performance VLSI Multiprocessor, Unpublished Master's Report, University of California, Berkeley (August, 1986).

[Emer84] J. S. Emer and D. W. Clark, A Characterization of Processor Performance in the VAX-11/780, *Proc. Eleventh International Symposium on Computer Architecture*, Ann Arbor, MI (June 1984).

[Fuji87] Fujitsu, Advanced Information of MB86900 -- A High Performance 32-bit RISC Processor (July, 1987).

[Gabr85] R. P. Gabriel, *Performance and Evaluation of Lisp Systems*, MIT Press, (1985).

[Gibs87] G. Gibson, Performance Estimates of Shared Memory Multiprocessors, Computer Science Division Technical Report UCB/Computer Science Dpt. 87/355, University of California, Berkeley (May 1987).

[Good83] J. R. Goodman, Using Cache Memory to Reduce Processor-Memory Traffic, *Proc. Tenth International Symposium on Computer Architecture*, Stockholm, Sweden (June 1983), 124-131.

[Groh82] G. F. Grohoski and J. H. Patel, A Performance Model for Instruction Prefetch in Pipelined Instruction Sets, *Proc. International Conference on Parallel Processing* (August 1982).

[Henn84] J. L. Hennessy, VLSI Processor Architecture, *IEEE Trans. on Computers*, C-33, 12 (Dec 1984).

[Hill84] M. D. Hill and A. J. Smith, Experimental Evaluation of On-Chip Microprocessor Cache Memories, *Proc. Eleventh International Symposium on Computer Architecture*, Ann Arbor, MI (June 1984).

[Holg80] R. W. Holgate and R. N. Ibbett, An Analysis of Instruction-Fetching Strategies in Pipelined Computers, *IEEE Trans. on Computers*, C-29, 4 (April 1980), 325-329.

[Joob85] R. Joobbami, WEAVER: An Application of Knowledge-Based Expert Systems to Detailed Routing of VLSI Chips, Ph.D. Dissertation, Department of Electrical and Computer Engineering, Carnegie-Mellon University (July, 1985).

[Joup86] N. Jouppi, *Private Communication*, Dec. Western Research Lab, (December 1986).

[Kate83] M. G. H. Katevenis, R. W. Sherburne, D. A. Patterson and C. H. Séquin, The RISC II Micro-Architecture, *Proc. VLSI 83 Conference*, Trondheim, Norway (August 1983).

[Katz85] R. H. Katz, editor. Proc. of CS292i: Implementation of VLSI Systems, Computer Science Division Technical Report UCB/Computer Science Dpt. 86/259, University of California, Berkeley (September 1985).

[Lee84] J. K. F. Lee and A. J. Smith, Branch Prediction Strategies and Branch Target Buffer Design, *Computer*, 17, 1 (January, 1984), 6 - 22.

[Lee87] D. Lee, *Private Communication*, U.C. Berkeley, (June 1987).

[Low87] C. Low and L. D. Rugg, Design of Branch Target Buffer and Perforamnce Comparison Against Instruction Cache, Unpublished CS 252 Class Project, University of California, Berkeley (May 1987).

[McFa86] S. McFarling and J. Hennessy, Reducing the Cost of Branches, *Proc. Thirteenth International Symposium on Computer Architecture*, Tokyo, Japan (June 1986).

[McLe82] H. R. McLellan, Jr., Instruction Prefetch Strategies in a Pipelined Processor, Unpublished Masters Thesis, M.I.T. (circa 1982).

[Moto84] Motorola Semiconductors, MC68020 Technical Summary, No. BR-243 (1984).

[Moto86] Motorola Semiconductors, MC68030 Technical Summary, No. BR508/D (1986).

[Mous86] J. Moussouris, L. Crudele, D. Freitas, C. Hansen, E. Hudson, R. March, S. Przybylski, T. Riordan, C. Rowen and D. V. Hof, A CMOS RISC Processor with Integrated System Functions, *Proc. Compcon* (Spring 1986).

[Patt82] D. A. Patterson and C. H. Séquin, A VLSI RISC, *Computer*, 15, 9 (September 1982), 8-21.

[Patt87] D. A. Patterson, *Private Communication*, University of California, Berkeley, (October 1987).

[Rau77] B. R. Rau and G. Rossmann, The Effect of Instruction Fetch Strategies Upon the Performance of Pipelined Instruction Units, *Proc. Fourth Symposium on Computer Architecture* (June 1977), 80-89.

[Siew82] D. P. Siewiorek, C. G. Bell and A. Newell, Computer Structures: Principles and Examples, McGraw Hill (1982).

[Smit78] A. J. Smith, Sequential Program Prefetching in Memory Hierarchies, *Computer*, 11, 12 (December 1978), 7-21.

[Smit82] A. J. Smith, Cache Memories, *Computing Surveys*, 14, 3 (September, 1982), 473 - 530.

[Smit83] J. E. Smith and J. R. Goodman, A Study of Instruction Cache Organizations and Replacement Policies, *Proc. Tenth International Symposium on Computer Architecture*, Stockholm, Sweden (June 1983), 132-137.

[Smit85] A. J. Smith, Cache Evaluation and the Impact of Workload Choice, *Proc. Twelfth International Symposium on Computer Architecture* (June 1985).

[Smit87] A. J. Smith, Line (Block) Size Choice for CPU Caches, *IEEE Trans. on Computers*, C-36, 9 (September 1987).

[Tayl87] G. Taylor, Design and Evaluation of the SPUR Lisp Architecture, Ph.D. Thesis, University of California, Berkeley (expected Fall 1987).

[Term83] C. J. Terman, Simulation Tools for Digital LSI Design, Laboratory for Computer Science Technical Report #304, MIT (September, 1983).

[Whol85] S. Wholey, S. E. Fahlman and J. Ginder, Revised Internal Design of Spice Lisp, Department of Computer Science Technical Report, Carnegie-Mellon University (January, 1985).

[Zorn87] B. Zorn, P. Hilfinger, K. Ho and J. Larus, Internal Design of the SPUR Lisp System, Computer Science Division Technical Report UCB/Computer Science Dpt. 87/373, University of California, Berkeley (September 1987).