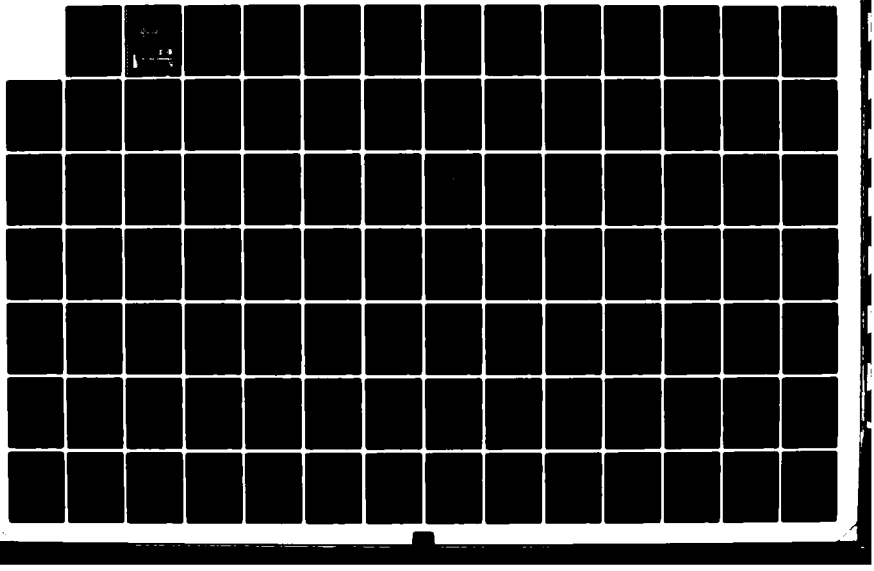
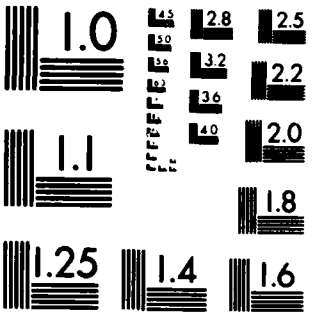
AD-A124 387 SHARED CACHE ORGANIZATION FOR MULTIPLE-STREAM COMPUTER SYSTEMS(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE LAB C YEH JAN 81 R-904 N00039-80-C-0556 1/3

UNCLASSIFIED F/G 9/2 NL



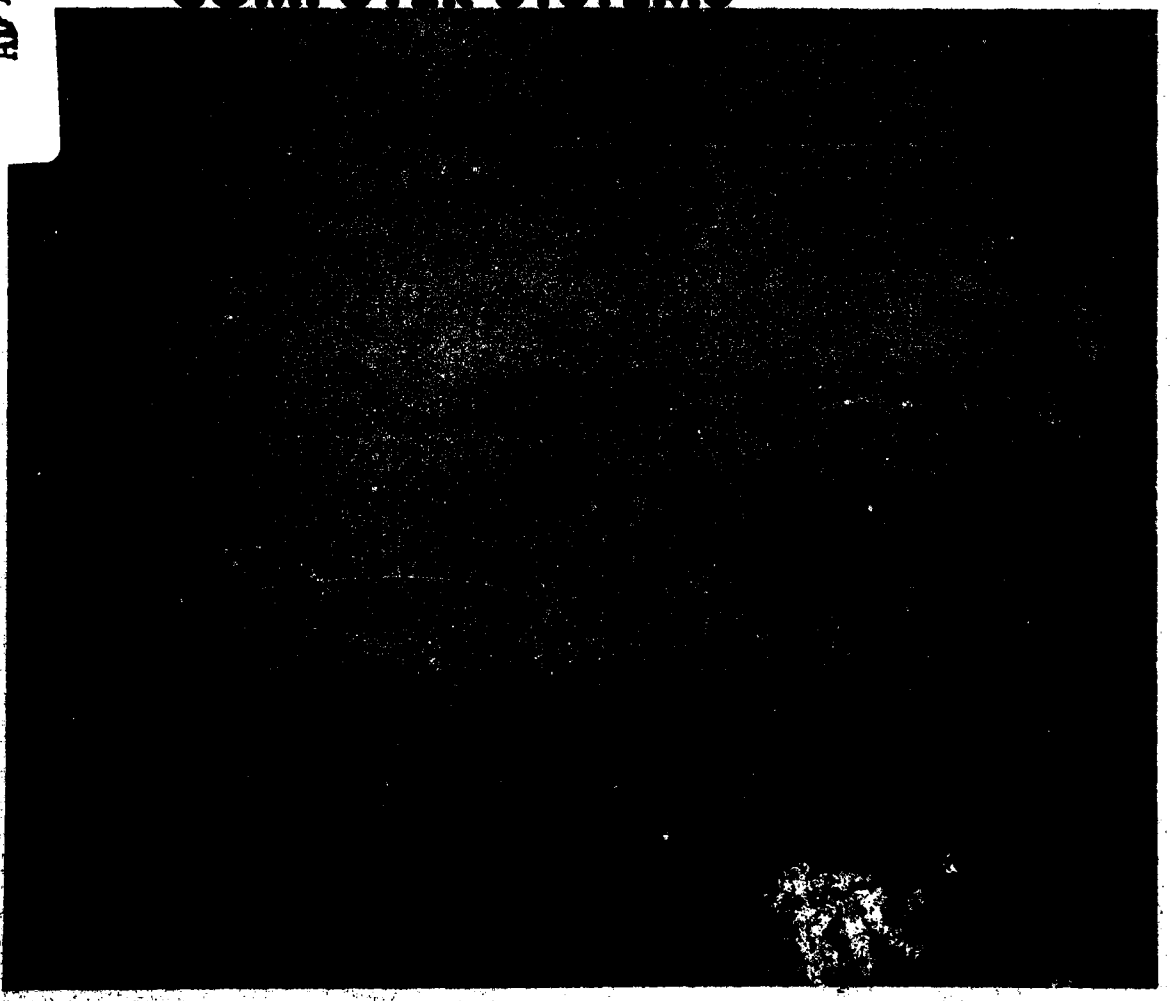


REPORT R-904 JANUARY, 1981 UILU-ENG 81-2236

(12)

CSL COORDINATED SCIENCE LABORATORY

SHARED CACHE ORGANIZATION FOR MULTIPLE-STREAM COMPUTER SYSTEMS



UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A124 387	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SHARED CACHE ORGANIZATION FOR MULTIPLE-STREAM COMPUTER SYSTEMS		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER R-904; UILU-ENG 81-2235
7. AUTHOR(s) CHI-CHUNG YEH		8. CONTRACT OR GRANT NUMBER(s) N00039-80-C-0556 N00014-79-C-0424
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS VHSIC Systems, US Navy Joint Services Electronics Program		12. REPORT DATE January, 1981
		13. NUMBER OF PAGES 260
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Cache memories, parallel memories, pipeline processors, parallel processors, multiprocessors, memory interference, performance evaluation, simulation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Organizations of shared two-level memory hierarchies for parallel-pipelined multiple instruction stream processors are studied. The multicopy of data problems are totally eliminated by sharing the caches. All memory modules are assumed to be identical and cache addresses are interleaved by sets. For a parallel-pipelined processor of order (s,p), which consists of p parallel processors each of which is a pipelined processor with degree of multiprogramming, s, there can be up to sp cache

DD FORM 1 JAN 73 1473

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

requests from distinct instruction streams in each instruction cycle. The cache memory interference and shared cache hit ratio in such systems are investigated.

The study shows that the set associative mapping mechanism, the write through with buffering updating scheme and the no write allocation block fetch strategy are suitable for shared cache systems. However, for private cache systems, the write back with buffereing updating scheme and the write allocation block fetch strategy are considered in this thesis.

Performance analysis is carried out by using discrete Markov Chain and probability based theorems. Performance is evaluated as a function of the hit ratio, h, the processor order, (s,p), and the cache organization characterized by the number of lines, ℓ, the number of modules per line, m, cache cycle time, c, and the block transfer time, T. Results shows that for reasonably large ℓ high performance can be obtained for shared cache with small (1-h)T. Shared-cache systems may perform better than private-cache systems if shared cache results in a higher hit ratio than private cache. The shared-cache memory organization is suitable for single pipelined processor systems because of the low access interference. Access interference of shared cache systems may be reduced to extremely low levels with a reasonable choice of system parameters.

Some design tradeoffs are discussed and examples are given to illustrate a wide variety of design options that can be obtained. Performance differences due to alternative architectures are also shown by a performance comparison between shared cache and private cache for a wide range of parameters.

SHARED CACHE ORGANIZATION FOR MULTIPLE-STREAM COMPUTER SYSTEMS

BY

Chi-Chung Yeh

B. Eng., Chung Yuan Christian College of Science and Engineering, 1972
M.S., Northwestern University, 1975
M.S., University of Illinois, 1977

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1981

Urbana, Illinois

SHARED CACHE ORGANIZATIONS FOR MULTIPLE-STREAM COMPUTER SYSTEMS

Chi-Chung Yeh, Ph.D.
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1981

Organizations of shared two-level memory hierarchies for parallel - pipelined multiple instruction stream processors are studied. The multicopy of data problems are totally eliminated by sharing the caches. All memory modules are assumed to be identical and cache addresses are interleaved by sets. For a parallel - pipelined processor of order (s,p), which consists of p parallel processors each of which is a pipelined processor with degree of multiprogramming, s, there can be up to sp cache requests from distinct instruction streams in each instruction cycle. The cache memory interference and shared cache hit ratio in such systems are investigated.

The study shows that the set associative mapping mechanism, the write through with buffering updating scheme and the no write allocation block fetch strategy are suitable for shared cache systems. However, for private cache systems, the write back with buffering updating scheme and the write allocation block fetch strategy are considered in this thesis.

Performance analysis is carried out by using discrete Markov Chain and probability based theorems. Performance is evaluated as a function of the hit ratio, h, the processor order, (s,p), and the cache organization characterized by the number of lines, ℓ , the number of modules per line, m, cache cycle time, c, and the block transfer time, T. Results shows that for reasonably large ℓ high performance can be obtained for shared cache with small (1-h)T. Shared-cache systems may perform better than private-cache systems if shared cache results in a higher hit ratio than private cache. The shared-cache memory organization is suitable for single pipelined processor systems because of the low access interference. Access interference of shared cache systems may be reduced to extremely low levels with a reasonable choice of system parameters.

Some design tradeoffs are discussed and examples are given to illustrate a wide variety of design options that can be obtained. Performance differences due to alternative architectures are also shown by a performance comparison between shared cache and private cache for a wide range of parameters.

ACKNOWLEDGMENT

The author wishes to express his deepest gratitude to his advisors, Professor Edward S. Davidson and Professor Janak H. Patel, for their patient guidance, helpful suggestions and invaluable friendship. The author would also like to thank Professors B. R. Rau, J. A. Abraham and M. Schlansker for their constructive discussions.

The author also wishes to thank his colleagues at the Coordinated Science Laboratory; Joel Emer, Alan Gant, Larry Hanes, David Yen, Tim Chou and Andy Pleszkun, for providing an intellectually stimulating environment.

Finally, the author is grateful to his wife, Grace, for her love and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

BACKGROUND AND MOTIVATION

1.1 Introduction

Despite advances in modern computer design, there will always be a need for machines more powerful than those currently available. Although performance may be improved by increasing the switching speed of the electronic components, to achieve even faster computation, we must also take new approaches that do not depend on breakthroughs in device technology, but rather on imaginative computer architecture design. Two architectural techniques, parallel computing and pipelined computing [1], can be employed to enhance the throughput of a computer system. Parallelism in various forms has appeared in several computers and has proved to be an effective approach to performance improvement. In some highly parallel computer systems, like the C.mmp system at Carnegie-Mellon University [2] and the AMP-1 system at University of Illinois [3], concurrency is achieved by a multiplicity of independent processors which execute separate instruction streams on separate data streams. This kind of system is referred to as Multiple Instruction Stream - Multiple Data Stream (MIMD) [4]. On the other hand, some highly parallel computer systems, like ILLIAC IV [5], STARAN [6], and PEPE [7], contain a large number of processors that perform the same computation on

a large collection of related data streams simultaneously. These are referred to as Single Instruction Stream - Multiple Data Stream (SIMD) [4].

Pipelining is one form of imbedding parallelism or concurrency in a computer system. A pipelined processor consists of several specialized subprocessors called segments. Each segment performs a specific part of a particular computation and operates concurrently with other segments. Instruction stream pipelining has been successfully implemented in many computer systems, such as IBM 360/91 [8] and Amdahl 470 V/6 [9], to overlap the instruction execution. Another form of pipelining is the data stream pipelining which performs the same arithmetic operation in an overlapped fashion on a series of operands as they flow through the pipe. Examples of these are the vector processors: CDC STAR-100 [10], TI-ASC [11], and CRAY-1 [12]. Pipelined processors appear to have an attractive architecture for multiprocessing systems in the near future because of their inherent cost advantage [13,14], regular structure and high pin utilization [15] which are very suitable for VLSI technologies. A general model and formal description of such a highly concurrent processor organization is presented in the next section.

Despite the significant progress in semiconductor technology now occurring, faster and larger storage will always be in demand. It has generally been recognized that these demands cannot be fulfilled at an acceptable cost with any single current technology, but this need can be satisfied by a memory hierarchy which combines a variety of technologies with differing cost-performance characteristics. Today, some memory

hierarchy is used in almost every modern computer system from microcomputers to large scale supercomputers. In this research, a two level cache-main memory hierarchy for a multiprocessor system is studied. This research is not particularly concerned with main-secondary hierarchies or uniprocessor systems.

In a tightly-coupled multiprocessor environment, main memory is a prime system resource which is usually shared by all the processors. However, cache memory is generally not shared among processors. When cache memory is used, a separate cache is usually attached to each processor [16]. In such systems, interprocessor communications are usually required since no processor can directly address another processor's cache memory. This kind of cache memory organization, referred to as private cache, causes the well-known multicopy of data problem (or coherence problem [17]) which means more than one nonidentical and inconsistent copy of data exists in the system. Generally speaking, a memory hierarchy has such a coherence problem as soon as one of its levels is split into several independent units which are not equally accessable from faster levels or processors.

In this thesis, a new solution is proposed which eliminates such coherence problems by sharing the cache memories among all processors. However, care must be taken in the organization of the shared cache memory system to avoid severe performance degradation due to cache memory access interference caused by two or more processors simultaneously attempting to access the same module or resoure of the shared cache memory system. A simple way to reduce access interference is to divide

the shared cache memory system into several independent modules. Then several cache requests would be able to access the cache memory simultaneously if they all reference distinct modules. Interleaving of the addresses among shared cache memory modules is then used to alleviate the interference problem.

Memory hierarchy is a cost-effective approach to obtain a balance between effective processor and storage cycles. The processor and cache memory cycles and the data transfer rate between various memory levels are all significant considerations for achieving balanced system design. The data transfer rate is in turn tightly related to the cache memory organization. The cache memory system is usually organized to meet the cache memory bandwidth requirements of the system. The memory bandwidth is the rate at which memory can provide information and is usually measured as words per second.

In this research, we characterize a wide variety of shared-cache multiple-stream computer systems and describe a method for evaluating their performance. Furthermore, the access interference problem, dynamic space-sharing phenomena and various cache memory control mechanisms are characterized and evaluated.

1.2 Processor Organization

The concept and advantages of pipelined processors have been introduced in section 1.1. A general and detailed model which describes

a multiple instruction stream multiple data stream (MIMD) processor implemented through pipelining is discussed in this section. A formal definition of the pipelined processor model used in this research is given below.

Definition 1.2.1 A pipelined processor of order s is modeled as an ordered set of s segments $(s_1, s_2, \ldots, s_{s-1})$, each of which can simultaneously be processing a distinct step or phase of a distinct instruction. □

Once an instruction is initiated in the initial segment, it flows from segment to segment for its execution, where each segment performs a specific suboperation on a distinct phase of the instruction. It is considered that each segment has an output latch or register to help retain its autonomy. Figure 1.2.1 illustrates a pipelined processor of order 4.

If successively initiated instructions are always taken from a single instruction stream, the processor is called a single instruction stream pipelined processor (or sometimes an overlapped machine). However, allowing successively initiated instructions to be interleaved from distinct instruction streams permits a single pipeline to implement a multiple instruction stream pipelined processor. The following definition aids in the understanding these two implementations.

Definition 1.2.2 The rth process or instruction stream, I(r) is a

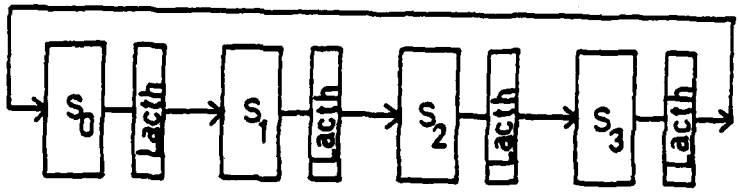


Figure 1.2.1 A pipelined processor of order 4.

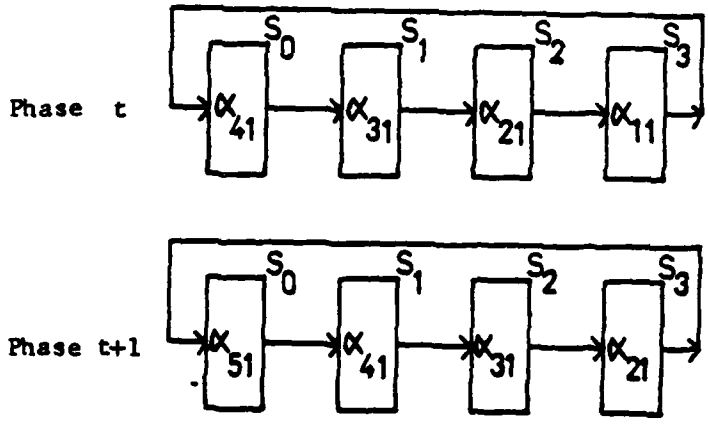


Figure 1.2.2 A single-stream, four-segment pipelined processor.

sequence of instructions that require execution. Thus,

$$I(r) = \alpha_{1r}, \alpha_{2r}, \alpha_{3r}, \cdots$$

where α_{ij} = ith instruction from the jth instruction stream. □

Figure 1.2.2 shows a single-stream, four-segment pipelined processor. In this scheme, execution of instructions from a single instruction stream are overlapped. The problems usually associated with single instruction stream pipelined processors are the performance degradation and control problems due to data dependencies and branch instructions. In this research, we restrict our attention to multiple instruction stream pipelined processor organizations in which the performance degradation and control problems due to data dependencies and branch instructions are absent. Figure 1.2.3 shows a four-stream, four-segment pipelined processor. In general, s distinct streams are in execution concurrently and if an instruction from a stream is initiated at time instant t, the next instruction from the same stream will be initiated at time instant t+s. Therefore, instruction execution overlap is achieved only between distinct instruction streams and no execution overlap occurs between instructions from the same stream.

The pipelined processor can be partitioned so that all segments take the same time to complete their execution phases. Then in a pipelined processor of order s, s seperate instructions will be in different phases of their execution at any time instant. Since these s instructions come from distinct instruction streams, the degree of multiprogramming is also s.

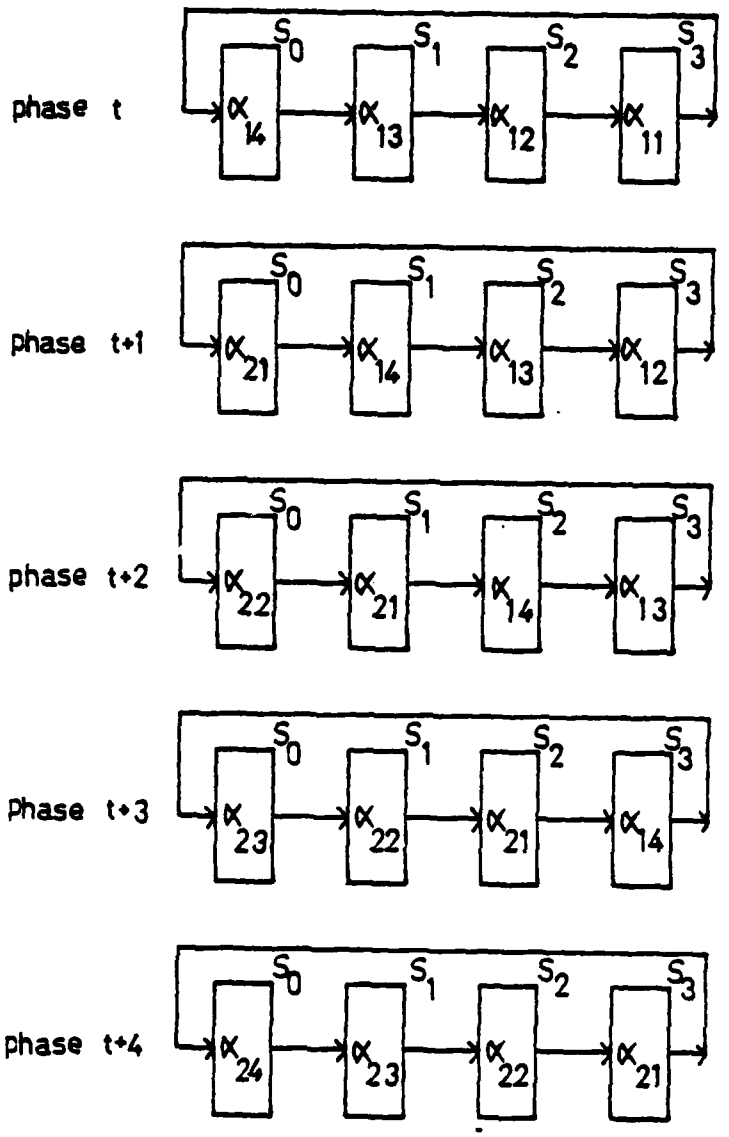


Figure 1.2.3 A four-stream, four-segment pipelined processor.

Definition 1.2.3 One segment time unit (STU), is the time, in seconds required by a segment to execute its distinct phase of an instruction. □

Hence if the phases of an instruction are partitioned so that it takes τ seconds to execute each phase of the instruction, then one STU is equal to τ seconds.

Pipelines in which all instructions have identical flow patterns are termed single function pipelines. On the other hand, in a multifunction pipeline, there are two or more distinct flow patterns and each instruction may use one of these flow patterns [18]. In this research, it is assumed that the pipeline processor is a single function pipeline.

Assume that each instruction can issue one memory request per instruction cycle (or pipelined processor cycle); where one instruction cycle = s τ seconds. Hence a pipelined processor of order s can issue one memory request per STU and a total of s requests can be issued in one instruction cycle. The instruction here is the unit instruction defined by Strecker [19] such that each instructon issues one memory request per instruction cycle and the instruction cycle is fixed. A pipelined processor is then completely characterized for our purposes by s, the degree of multiprogramming, and τ, the segment time.

The generalized processor organization is now discussed.

Definition 1.2.4 A parallel-pipelined processor of order (s,p) [20] is modeled as a set of p identical and independent, but synchronized processors, each of which is a pipelined processor of order s. □

Figure 1.2.4 illustrates the possible configurations of parallel-pipelined processors. A parallel-pipelined processor is thus completely specified for our purposes by the degree of multiprogramming, s, the parallelism, p, and the segment time unit, τ . A parallel-pipelined processor of order (s,p) executes sp distinct instruction streams concurrently and issues p simultaneous memory requests per STU. From now on, all time units will be expressed as an integer number of STUs, unless otherwise stated.

1.3 Program Behavior and Memory Hierarchies

Two types of referencing behavior have been found to be characteristic of almost all programs: temporal locality and spatial locality [21]. Temporal locality implies a higher probability of referencing information used more recently than that referenced a long time ago. A high degree of temporal locality is expected from programs with loops. Spatial locality implies a high probability of making references in the near future to information which is close (in the logical address space) to recently referenced information. We should expect that programs will execute code sequentially and when branches do occur they are usually over short forward or backward distances. Sequentiality is a specific form of spatial locality. The principle of sequentiality indicates that the successive information following the information currently accessed is likely to be referenced next. This type of behavior is expected from common knowledge of programs, i.e.

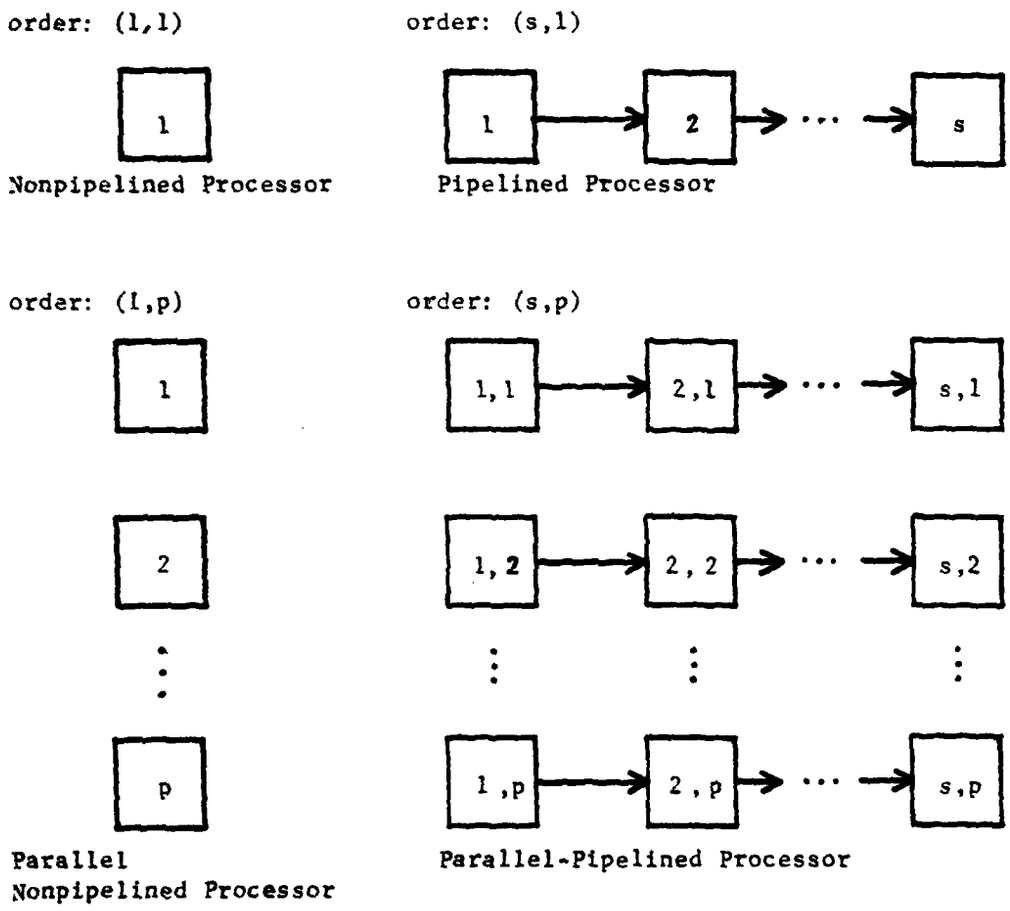


Figure 1.2.4 Configurations of parallel-pipelined processors.

related data items (variables, arrays) are usually stored together, and instructions are often executed sequentially and input/output files are usually accessed sequentially. Substantial sequentiality can also be seen in data-base systems [22].

Some degree of temporal locality and spatial locality is inherent in all programs. Their existence makes it worthwhile to retain in fast access storage a subset of all the information which has been referenced in the near past. On the other hand, prefetch or block fetch is used to improve system efficiency by predicting the spatial locality or sequentiality.

The sequentiality and locality of referencing behavior, commonly found in the memory referencing patterns of computer programs, can be used to predict which sections of a program's address space are likely to be referenced next. Due to program locality and sequentiality, memory hierarchy is a cost-effective approach to improving the effective storage access cycle by prefetching information from slower to faster memory levels before the information is actually accessed and by retaining frequently used information in the fastest memory level. Under temporal and spatial locality, memory hierarchies attempt to maximize the probability that information is in the faster storages when being referenced. Memory hierarchies then achieve the approximate speed of small, fast storages while maintaining the approximate cost-per-bit of the larger, slower storages with lower cost per bit. In this section, some program behavior is discussed and a general description of memory hierarchies is presented.

In general, an H-level paged memory hierarchy consists of a collection of memory devices M_1, M_2,..., M_H, a network of paths connecting the devices, and a hierarchy management facility [23]. Each device is partitioned into physical blocks called pages. The hierarchy management facility controls the page movement between the devices. A reference from the processor can usually be serviced only from the highest storage level, M_1. Thus if the desired page resides in a lower level storage M_i, where $i > 1$, the hierarchy management facility must bring that page up to M_1 for serving a request. Figure 1.3.1 illustrates an H-level storage hierarchy. A storage hierarchy is called a linear storage hierarchy if the only paths for moving pages down the hierarchy are direct paths from each level, M_i to the next lower level, M_{i+1}, where i=1, 2,..., H-1. Since we only consider a two-level cache-main memory hierarchy, our memory hierarchy is a linear storage hierarchy.

For a two-level cache-main memory hierarchy, information is usually fetched to cache memory on a demand basis whereby, when a datum is referenced and is found to be absent from the cache (called a miss), it is copied from the main memory to cache memory. On the other hand, a reference is called a hit if the desired datum is found in the cache memory. The miss ratio is the fraction of all cache memory references resulting in a miss. Similarly, the hit ratio is the fraction of all cache memory references resulting in a hit.

The locality property of program behavior is usually considered during hierarchy management design in an attempt to maximize the hit ratios. For example, in the case of a paged main-secondary memory

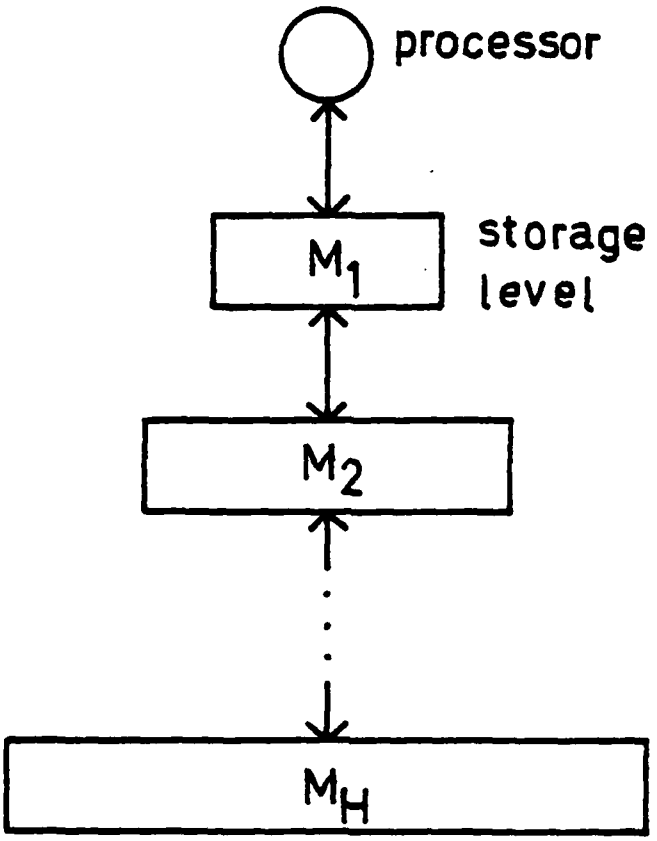


Figure 1.3.1. An H-level storage hierarchy.

hierarchy, the working set replacement scheme [24] performs well because it takes advantage of program locaility. However, it is known that many factors can affect the performance of a memory hierarchy. In addition to program behavior, performance is also a complicated function of each memory level organization, the capacity and cycle time of each memory device. A more careful and detailed study of shared cache-main memory hierarchy will is presented in chapter 2.

1.4 Characteristics of Cache Memory Devices

In order to achieve the speed requirement of a cache memory system design, semiconductor memories are usually employed. Two types of semiconductor memories, namely Random Access Memory (RAM) and Content Addressable Memory (CAM) are normally used in the synthesis of cache memory systems. With today's semiconductor technology, RAM chips can provide high-speed operation comparable to the processor speed.

However, due to the inherent necessity of mapping pages among the levels in a memory hierarchy, intensive searches are usually executed for each memory reference to determine the physical location of the desired information. Searching is time-consuming in a RAM system because serial searching must be employed. CAM devices provide capability for parallel searching which allows mapping table lookup in one memory cycle. To obtain a high-performance cache memory system, mapping information is often stored in an associative lookup table (or cache directoty) in CAM devices rather than RAM devices. In this section, the characteristics of

both RAM and CAM chips are discussed.

Typically, RAM chips consist of five functional blocks as shown in figure 1.4.1. They include an address register and decoder, the storage cells, and the input and output buffers. In some memory chips, the address register and the output data buffer are not fabricated on the memory chip. However, with recent developments in LSI technology, the cost of fabricating these buffers on the memory chip is insignificant.

Let t_{cs} and t_{ad} be the chip select pulse width and the address pulse width respectively. Furthermore, t_{di}, t_w and t_{do} denote the data input pulse width, the write enable pulse width and the data output pulse width, respectively. For simplicity, assume that the chip select and address signals are gated into the chip simultaneously. Simplified typical timing diagrams for the read cycle and the write cycle of the RAM chips are shown in figure 1.4.2. and figure 1.4.3, respectively.

Definition 1.4.1 The memory cycle, t_c, is the time that a memory chip remains busy after a memory operation is initiated. For a read memory operation, the cycle is called the read memory cycle, t_{rc}. Similarly, for a write operation, it is called the write memory cycle, t_{wc}. □

Definition 1.4.2 The memory access time, t_{ac}, is the time duration between a memory operation being initiated and the output data becoming available. □

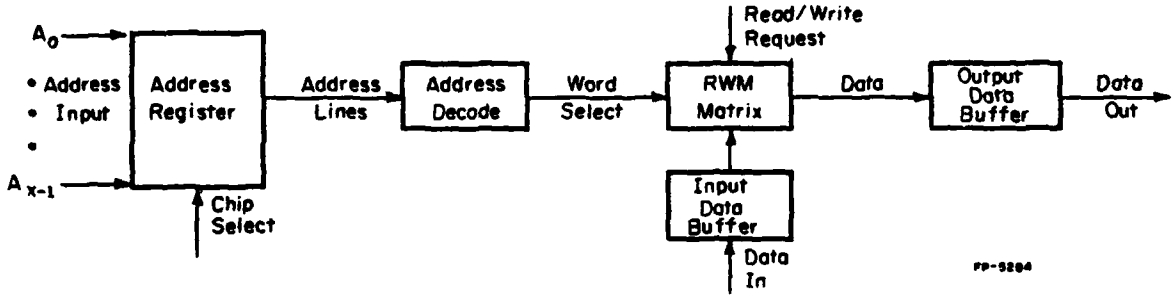


Figure 1.4.1 Functional blocks of RAM chip.

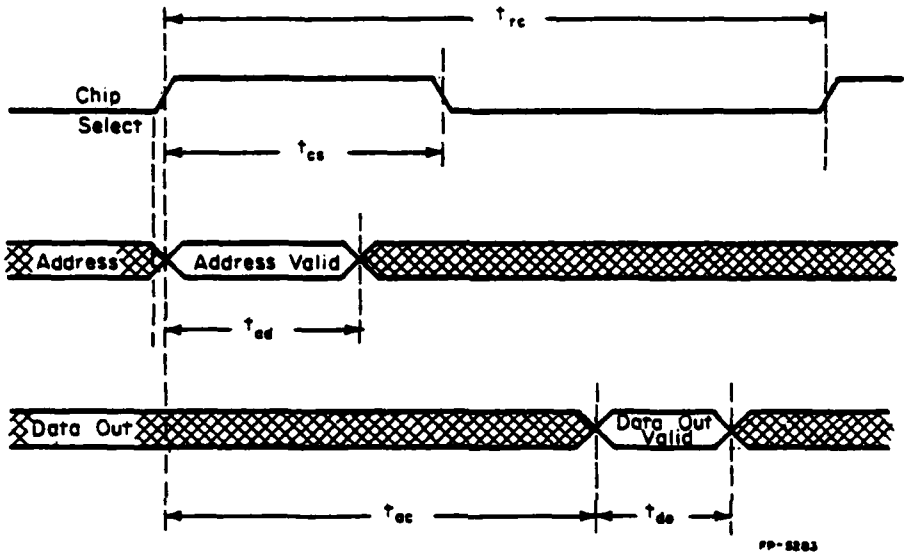


Figure 1.4.2 Timing diagram for read cycle of RAM chip.

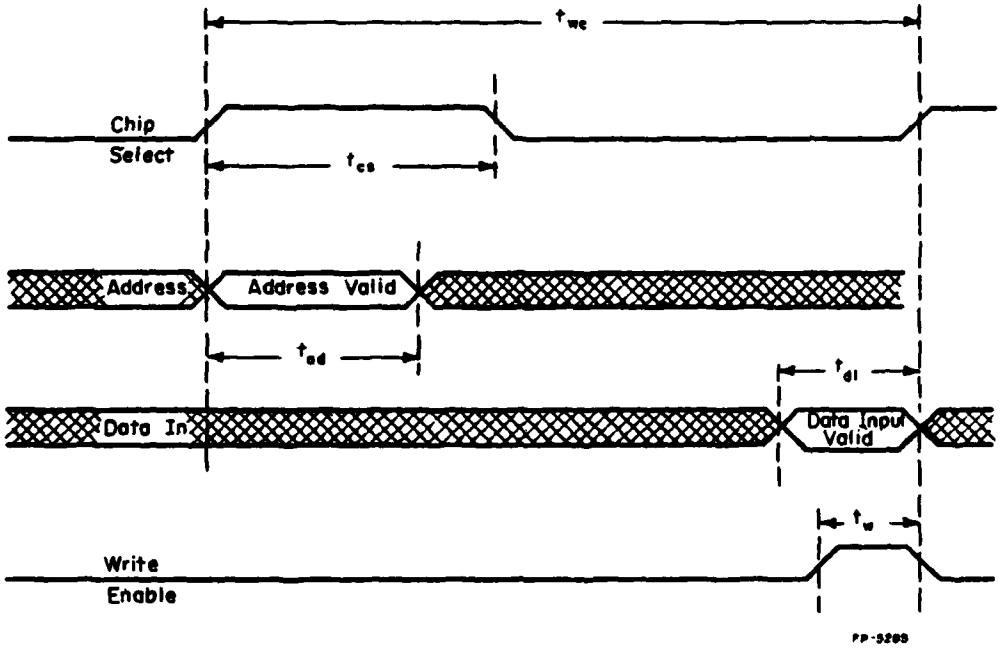


Figure 1.4.3 Timing diagram for write cycle of RAM chip.

Definition 1.4.3 The address hold time, t_a, is the minimum time duration that the address must be maintained at the input to the memory chip for a successful memory operation. □

A typical CAM chip also consists of five functional blocks as shown in figure 1.4.4. These include the input/output buffers and mask register, the storage cells and select circuit. In a RAM the information selected for reading or writing is identified by means of an explicit address. However, in a CAM the selection is done on the basis of the contents of the storage cells. Usually, each unit of stored information is a fixed-length word. Any subfield of the word may be chosen as the key. The mask register selects bits to be compared and the key is pattern of 1 and 0 bits in selected key positions. The key is compared simultaneously with all stored words; those which match the key emit a match signal which enters a select circuit. The select circuit enables the data field of the selected word to be accessed. If several entries have the same key, then the select circuit determines which data field is to be accessed. Since all words in the memory are required to compare their keys with the input key simultaneously, each must have its own match circuit. The match and select circuits make CAM chips much more complex and expensive than RAM chips.

The logic circuit for a 1-bit CAM cell (bit j of word i) is shown in figure 1.4.5. It comprises a flip-flop, a match circuit for comparing the flip-flop contents to an external data bit, and circuits for reading from and writing into the cell. To write information into this cell, the write enable signal (WE) is set to 1, S_i (select) is set to 1 for word

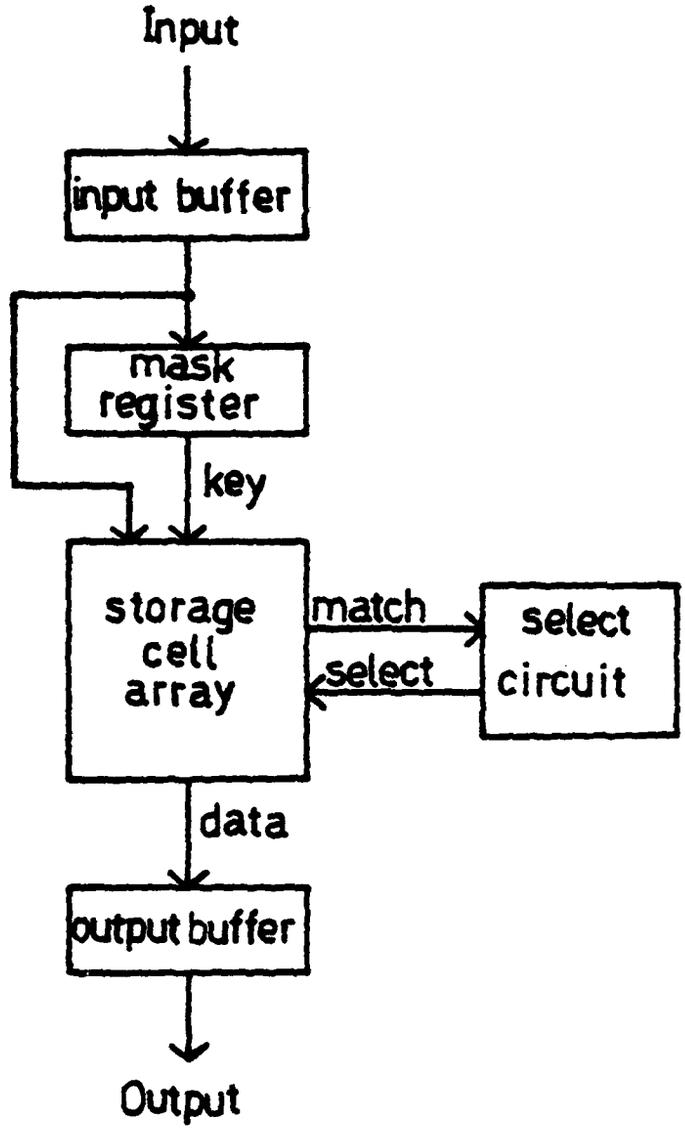
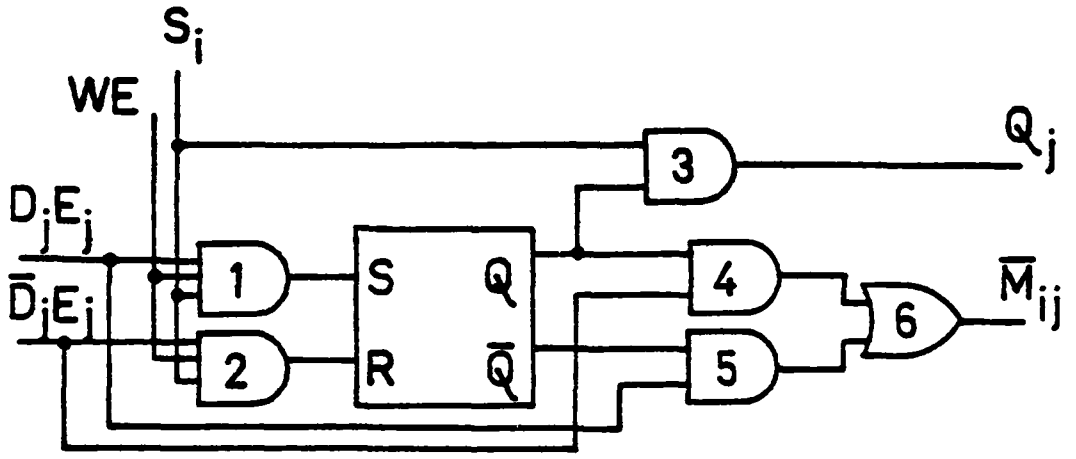


Figure 1.4.4 Functional blocks of CAM chips.



S_i = word i select
WE = write enable
E_j = data bit j enable
D_j = data bit j input
Q_j = data bit j output
M_{ij} = match for bit j of word i

Figure 1.4.5 A 1-bit CAM cell.

(i) into which writing is desired, E_j is set equal to 1 for all j, and D_j is set to the value of the data to be written. Reading is accomplished by setting WE=0 and S_j=1 for the desired word to be read. The word location contents will then appear on the output Q_j. To search for a match, E_j is set equal to 1 for those key positions which are to be matched. Bit positions for which E_j=0 will have match signals $\overline{M}_i$=0. The search key is entered into the D_j for the selected bits. Any cell for which there is a mismatch between D_j and Q_j will generate a 1 on $\overline{M}_{ij}$ if E_j=1, otherwise a 0 is generated on $\overline{M}_{ij}$. Figure 1.4.6 shows a 4 x 4 CAM memory array [25]. Note that the $\overline{M}$ lines of all cells in the same word are connected by a wired-OR gate. Similarly, the output lines of bit j in all words are connected by a wired-OR gate for each j. Since $\overline{M}_i$ is the OR of $\overline{M}_{ij}$ for all j, $\overline{M}_i$ will be 0 if and only if no $\overline{M}_{ij}$=1 for any j for which E_j=1. Thus $\overline{M}_i$=0 if and only if a match is discovered in all selected bit positions.

Foster [26] showed a ratio of 9:6 (or 7:5 if wired-OR is allowed) between the number of gates required to make a bit of CAM and a bit of RAM. Thus CAM's would cost between 1.4 and 1.5 times as much as RAM's if semiconductor prices were purely based on the number of gates required. However,this basis is not correct because, due to the economics of mass production of integrated circuits, prices can depend more on volume of production than on complexity of circuitry. Lamb [27] presents a price comparison based on the available commerical prices in July 1978. He showed that the price per bit of a 16-bit CAM chip (speed is 35 ns) offered by Intel Corp. can be 287 times that of a 1 K static RAM (45

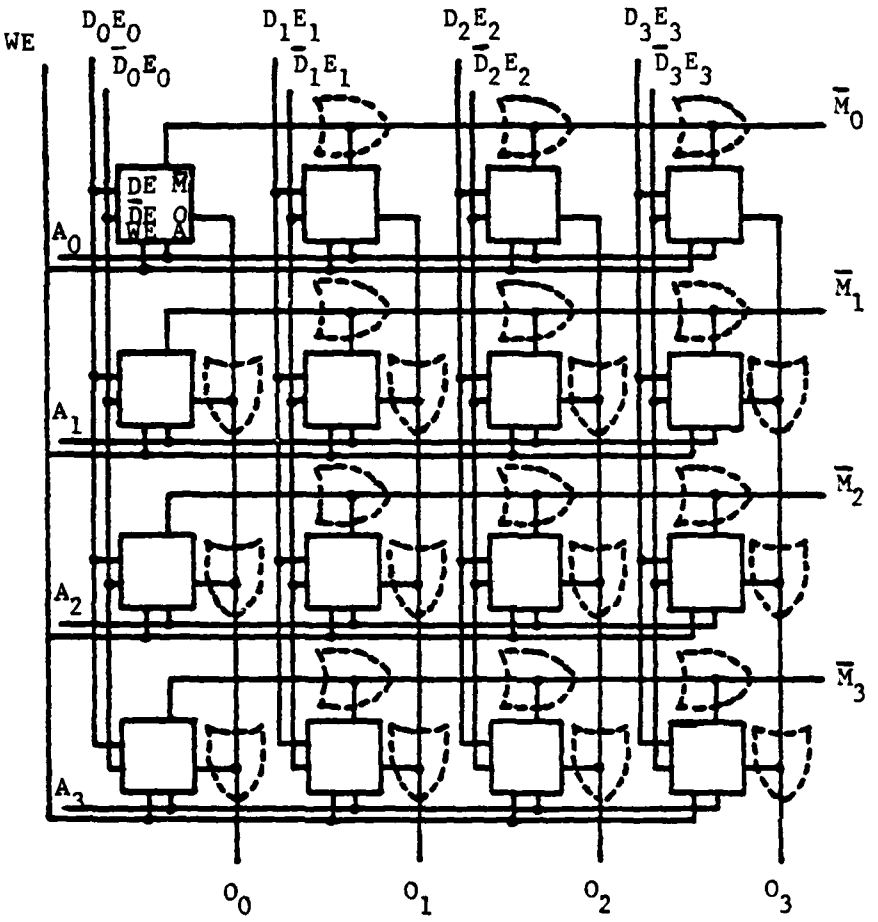


Figure 1.4.6 A 4 X 4 CAM memory array.

ns) offered by the same company. The cost of CAM chips is so expensive that a large lookup table implemented by CAM devices in a cache memory design is usually prohibitive. However, cache memories with a reasonable performance can be implemented by using RAM devices only. These alternatives will be discussed and modeled in the following chapters.

In general, $t_{rc} \leq t_{wc}$ for RAM chips. Since the read memory cycle, t_{rc}, may not always equal the write memory cycle, t_{wc}, an <u>effective memory cycle of the RAM chip</u>, which takes into consideration the distribution of read and write memory accesses is introduced. Assume that the fraction of read and write accesses of all memory requests are f_r and f_w, respectively, such that $f_r+f_w=1$. Then the effective memory cycle of the RAM chip is $t_{ec} = f_r\, t_{rc} + f_w\, t_{wc}$ For analytical purposes, "memory cycle" will mean the effective memory cycle and is used as the minimum time between two successive requests which can both be accepted by a particular memory module. For CAM chips, the difference between t_{rc} and t_{wc} is insignificant and sometimes $t_{rc} = t_{wc}$, thus $t_{ec} = t_{rc} = t_{wc}$ is assumed.

Therefore, a memory device is characterized by its cycle time, t_c. This is referred to as the <u>absolute</u> memory cycle because the cycle, t_c is expressed in seconds. However, the memory cycle can be quantized as an integer number of STUs, namely $c=\lceil t_c/\tau \rceil$, where τ is the segment time unit in seconds. Hence the <u>relative</u> memory cycle is c.

1.5 Objectives of This Research

In general, a private-cache memory is attached to each processor in a tightly-coupled multiprocessor computer system to improve the system efficiency. The typical structure of these systems is illustrated in figure 1.5.1. However, such a system will have the multicopy of data problem as mentioned in section 1.1. Note that reentrant (or pure) code avoids the multicopy of code problem because no modification of code is allowed. This coherence problem usually exists in the following three distinct forms:

(1) Multiple copies of shared data may exist in several private-cache memories. Modification of any shared data by a particular processor in its own cache memory will result in an obsolete value of this shared data in every other cache memory.

(2) Multiple copies of data may exist in several distinct memory levels. Modification of this data by a particular processor in its own cache memory will result in an obsolete value of this data in main memory. This difficulty may occur even in a uniprocessor with an independent I/O channel because I/O channels are normally connected to the main memory instead of to the cache memory. In this case, the most recently updated version of the data may be either in main memory or cache memory.

(3) In a multiprogramming system, a processor usually switches to

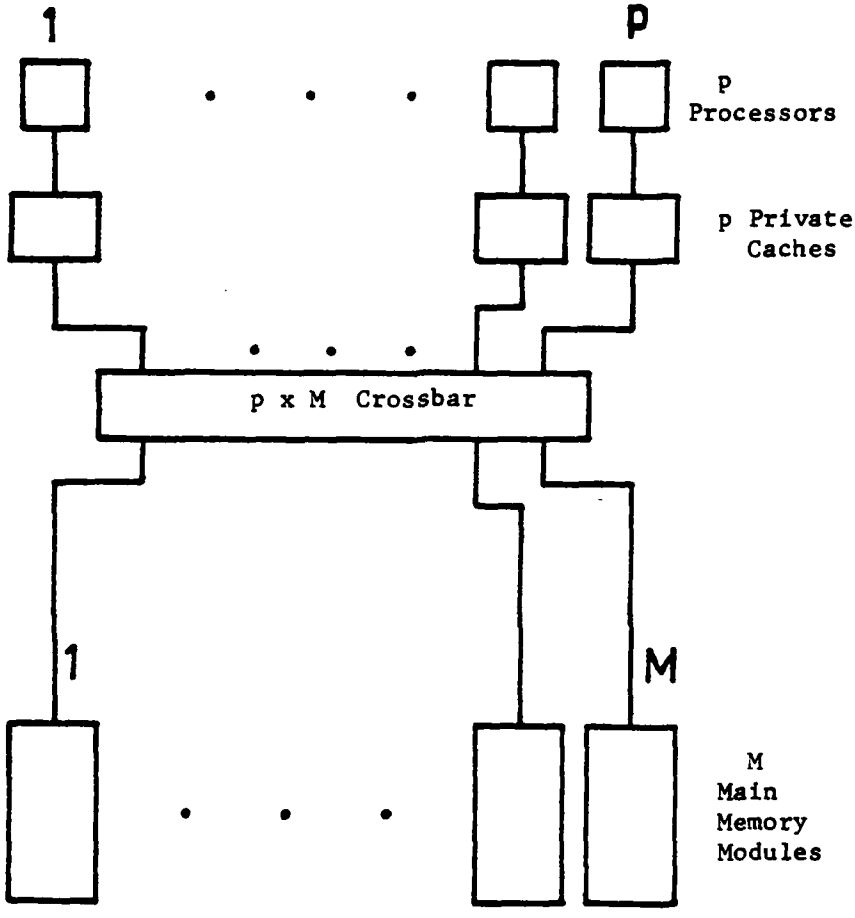


Figure 1.5.1 Multiprocessor system with private-cache memories.

other jobs at the time of arrival of external interrupt signals or input/output operations. After a job has been switched, the most recently updated data of this job might still be in the original processor's cache memory. Hence a job running on a new processor could use stale data in main memory. The new processor cannot recognize the data as stale, and thus would not be working with the job's proper context. Such operation is incorrect and can result in subtle errors that are difficult to trace down.

In addition to the problem of possible incorrect operation due to the multicopy of data, another problem is important in terms of availability. It may be the case that a processor has modified data in its cache for several different jobs before the main memory is updated. If the processor suffers a failure before the main memory is updated for those modified data, then other processors cannot select any of these jobs since their most recently updated data, stored in the cache associated with the original processor, cannot be directly accessed. Therefore, these jobs are effectively lost. The jobs must be manually restarted from the begining or from the last checkpoint. It is clear that coherence problems may occur even if no data is shared between jobs. Also this difficulty exists even in uniprocessor systems with an independent I/O channel.

In this research, a shared-cache memory structure is proposed to eliminate the multicopy of data problem for multiprocessor systems. This

solution can resolve all three difficulties mentioned above without any overhead penalty and hardware cost. However, with shared-cache memory there potentially are cache access conflict problems. Hence the purpose of this research is:

(1) To investigate the design methodology for shared-cache multiple-stream systems.

(2) To find proper cache management strategies for shared-cache memories.

(3) To study the effect of program characteristics on dynamic space-sharing.

(4) To evaluate the effect of cache memory interference on system performance for a variety of shared-cache memory configurations.

(5) To evaluate some design tradeoffs for obtaining cost effective shared-cache memory and memory hierarchy configurations.

1.6 Some Related Work

Although multiprogramming and time-sharing systems have been with us for a long time, very little work has been done on the effect of the interactions of various program characteristics on dynamic space-sharing. The working set model of program behavior has been extensively used to study these systems. Yet little attention has been given to dynamic

multiprogram interaction; only average or stationary characteristics have been investigated. Belady and Kuehner [28] studied an empirical model of the lifetime function, i.e. an average interval of program execution, for multiprogramming systems. Their conclusion showed that an increase of space does not significantly improve the processing potential when the space allocated to a task is small. For a large space allocation, the processing increment induced by additional space improves rapidly. Finally, when the task acquires a sufficiency of space, the processing improvement by adding more space is approximately zero. Note that the lifetime function is a stationary measurement. If the program's behavior during a subinterval can differ significantly from the average, conclusions based on the lifetime function may be inaccurate.

Coffman and Ryan [29] modeled the working-set size as a normal stochastic process and obtained more insightful results on the characteristics of dynamic space-sharing. Their general conclusion was that dynamic storage partitioning is superior when the variation in working-set sizes is relatively large. One common conclusion obtained by the above authors is that dynamic storage partitioning would not give worse performance than that of fixed storage partitioning. Dynamic storage partitioning performs better than fixed storage partitioning because space-sharing is superior if some processes have large working-set sizes while other processes have small working-set sizes. However, program working-set size is a function of time. Even with a large variation in working-set sizes, dynamic storage partitioning may not perform better than fixed storage partitioning if large working-set

sizes for some streams do not mostly match small working-set sizes for other streams in time.

Rodriguez-Rosell [30] observed the oscillatory pattern in working-set size behavior when the working set window is small. Vantilborgh [31] explained this dynamic behavior of the working set size by using a mathematical model. Unfortunately, they did not investigate the effect of interactions of the dynamic behavior of several streams on system performance.

Cache memory is usually so small that it cannot contain the entire working set of a single program. Hence, space contention in a shared cache memory is much more severe than that in a shared main memory. The dynamic interaction between several programs thus has a significant effect on the performance of shared cache systems. In this research, the phenomena of space-sharing under dynamically interacting programs of various kinds are investigated.

One important application of multiprocessor systems is the parallel multiprocessing environment, such as image processing and matrix computation. In such applications, many synchronization and communication operations between processes are needed and they are usually implemented with the aid of critical sections and semaphores. In addition to the possible computational data shared among processes, critical sections and semaphores also involve a form of sharing. Therefore, it is desirable to handle the coherence problem efficiently for a high performance system under a large shared data workload.

The difficulties caused by the multicopy of data problem in private-cache multiprocessor systems have been introduced in section 1.5. Some systems, like C.mmp [2], try to avoid this coherence problem by allowing only information from "read-only" pages (especially instructions) to appear in the cache. In other words, the "store algorithm" used in this system is the "store only in main memory" algorithm. However, the mean write rate for most processor architectures is between 10 to 30 percent of all accesses. For some instructions, the peak rate is much higher: 50 percent for a long move and 100 percent for a move immediate [17]. A commonly high write rate will greatly degrade the performance of a read-only cache system. In another solution, called the classical solution [17], addresses of modified blocks are broadcast throughout the cache memories for invalidation. To insure coherence, every cache is connected to a communication path over which the addresses of blocks to be modified are sent. Each cache constantly monitors this path and executes the proper operations for invalidation. Censier and Feautrier [17] point out that the drawbacks of this solution are: high invalidation traffic, low cache efficiency and the need for buffers to accommodate the peak invalidation traffic.

Recently, Tang[16] proposed an algorithm which includes a centralized "store controller" and a "central directory" to keep track of every block in each cache memory. Also, he assumed that the "store only in cache" algorithm is used. Each block is identified as shared or private. A shared block can have more than one copy existing in different caches, but allows read access only. A private block can have

only one copy in the caches at any time, but allows write access. In his solution, overhead is due not only to the extensive search operations executed in the store controller but also due to the checking of every desired block status in order to initiate a cache memory operation. As the shared data between tasks becomes large, normal cache operation may be interferred with by commands from the store controller to change shared block status. This interference and overhead may be severe for some kinds of shared blocks, such as those which contain critical sections or semaphores, for which the status may have to be changed back and forth many times during execution. Furthermore, the access conflict problem may occcur at the store controller because it is centralized and shared by all caches and channels. Censier and Feautrier [17] proposed a solution very similar to Tang's algorithm. In addition to the drawbacks of their solutions mentioned above, none of these solutions can efficiently resolve the coherence problem for multiprocessor systems in a multiprogramming environment. That is, when a processor wants to switch the job, the processor must sweep its cache to validate main memory before another processor can run the same job.

In this research, the coherence problem is resolved by the architectural approach of sharing caches. The potential performance degradation of this proposed structure is simply cache memory access interference. However, this cache memory access conflict problem can easily be overcome by using a sufficiently large number of cache modules and can theoretically be alleviated to any desired degree.

Various analytic and simulation models have been developed to study

memory access conflicts. Several models [32-35] seem to assume a single processor with instruction look-ahead capabilities. Here, we will only present the analytic models of interleaved memory in multiprocessors. The discrete Markov chain model proposed by Skinner and Asher [36] is limited to a small number of processors (≤ 2) because of the complexity involved for large systems. Strecker [19] investigated the conflict problem in a multiprocessor system with P processors and N memory modules. By approximate analysis, a closed form representation of the memory bandwidth was obtained as $N[1-(1-1/N)^P]$. Ravi[37] studied a similar model and derived a complicated solution for expected memory bandwidth. It is interesting to note that Strecker's formula is a closed form representation of Ravi's formula. Bhandarkar [38] expanded on Strecker's results. Sastry and Kain [39] had similar models but also investigated performance using distinct storage for instructions and data with interleaving only in the instruction space. Baskett and Smith [40] have also investigated the memory conflict problem in multiprocessor systems. They derived several approximate solutions and compared their predictions with simulation results. Briggs and Davidson [41,42] studied a more general multiprocessor model in which the system consists of a wide variety of parallel-pipelined processors of order (s,p) with two dimensional interleaved memory configurations. The other models cited involve special cases of their multiprocessor system models. In this thesis, an adaptation of Briggs and Davidson's memory organization will be used as our shared-cache memory organization. A more detailed review of their memory model is presented in chapter 2.

1.7 Overview of the Dissertation

Background material and motivation of the research have been presented in this chapter. In chapter 2, the shared-cache memory organization is discussed. A study of cache management strategies for the shared-cache memory is given. Total system configurations are also outlined. In chapter 3, the performance of the shared-cache memory system is analyzed for two distinct cache models. A discrete Markov approach and a probabilistic approach are developed for both models. A probabilistic model for private-cache systems is also evaluated. Bounds on performance are obtained for one of the shared-cache models. In chapter 4, the accuracy of these models is evaluated by simulation. Some effects of program behavior on dynamic space-sharing are discussed. The effects of the various parameters on performance are investigated. In addition, some design tradeoffs are studied. The performance of shared-cache systems are compared with that of private-cache systems. Chapter 5 presents overall conclusions and prospects for further research.

CHAPTER 2

SHARED CACHE MEMORY ORGANIZATION

2.1 Introduction

As mentioned in the previous chapter, the use of cache memory provides an effective memory access time at the system level near that of the fast smaller cache memory with the apparent memory capacity near that of the large and slower main memory. Thus, the processor ideally tends to operate with a memory of cache speed but with main memory cost-per-bit. This performance goal is similar to that of other systems using memory hierarchies, such as paging or virtual memory systems. However, there are some important differences between the cache-main memory hierarchy and the main-secondary memory hierarchy. In contrast with main-secondary memory hierarchy, a cache is managed by hardware rather than software, deals with smaller blocks of data, uses a smaller ratio of memory access times, accesses second level memory directly, and holds the processor idle rather than switching to another task while blocks of information are being transferred from main memory to cache. These important differences significantly affect the choice of design parameters for these distinct memory hierarchy systems. For example, in a multiprogramming paging system, the processor switches to another task when a page fault occurs. Task switching makes the page transfer time

less critical to the system throughput. Task switching is necessary since the ratio of memory access times can be as high as 1000:1 and the task switching overhead is far lower than the page miss wait time. Page hit ratio and task switching time rather than page transfer time are considered as important parameters in such a main-secondary memory design. However, in a cache system, the processor is forced to wait when a cache miss occurs. Thus the throughput of this system critically depends on not only the cache hit ratio but also the block transfer time.

Before any performance analysis for a cache system can be done, certain cache design parameters have to be determined in advance. In general, these parameters can be classified as functional (organizational) parameters and component parameters. Functional parameters determine the hardware functions and system organization in a cache system, such as address mapping mechanism, replacement algorithms, main memory updating schemes, and so on. Component parameters determine the physical sizes of various components in a cache system, such as total cache size, block size, cache memory cycle time, block transfer time, and so on. Although these two kinds of cache parameters are not independent of each other, from a system design point of view, functional parameters should be determined prior to component parameters. In this chapter, only the functional parameters of a shared cache memory design are discussed and determined. A range of component parameters is examined and their effects compared in chapter 4.

Cache memories have been with us for more than twenty years [43]. Today, cache memories are used by many of the prevalent machines (such as

IBM 360/85, 360/195, 370/158, 370/168, Amdahl 470V/6, DEC PDP 10/L, PDP 11/70, etc.). The performance of cache memory for uniprocessor computer systems is well-known [44,45,46-51] and satisfactory. But the design of a shared cache memory for multiple-stream computer systems is quite different from the design of a cache memory for conventional uniprocessor computer systems. Furtor considerations apply to shared-cache multiple-stream computer system design. Section 2.2 reviews various cache memory mapping mechanisms and discusses their feasible application to shared-cache memory design. In section 2.3, replacement algorithms and main memory updating schemes for shared-cache memory is discussed. Section 2.4 presents the L-M shared-cache memory organizations. Section 2.5 illustrates various memory interleaving implementations and their corresponding addressing formats. Section 2.6 explains request scheduling in a shared-cache memory system. The last section gives overall system configurations and considerations about realistic implementations of some hardware functions.

2.2 Cache Memory Mapping Mechanisms

The addresses assigned to a cache are maintained in a hardware-implemented memory map. Usually, this hardware memory map is implemented based on an associative memory. Associative memories are very expensive, so that a number of more economical memory mapping mechanisms have been proposed [52]. Besides the consideration of cost, several undesirable features may occur when some of these proposed

mechanisms are applied directly to a shared-cache memory design for multiple-stream computer systems. In order to explain these undesirable features and to determine a proper mechanism to be used in the future discussion, the previously proposed mapping mechanisms are examined below in detail.

2.2.1 Fully Associative Mapping

The conceptually simplest memory mapping scheme is called fully associative. Cache memory and main memory are logically divided into equal size blocks. In this case, any block from main memory may be mapped into any block in the cache as shown in figure 2.2.1(a). This requires that for each block stored in the cache, a corresponding block address must also be stored in the associative lookup table, and at the time of each processor request, a complete lookup table search must be made for the referenced address. A fully associative lookup in a cache tends to be extremely expensive and/or slow, because of the large search required. However from the space contention point of view the fully associative scheme is the theoretically optimum mechanism for uniprocessor systems. Unfortunately, this parallel lookup search becomes the most unacceptable drawback to using the fully associative map in a multiple-stream computer system. Recall that a parallel-pipelined processor of order (s,p) can issue p simultaneous cache memory requests each STU. Then p-1 requests out of those p requests will be rejected due to the conflict of parallel lookup search if one single-ported fully

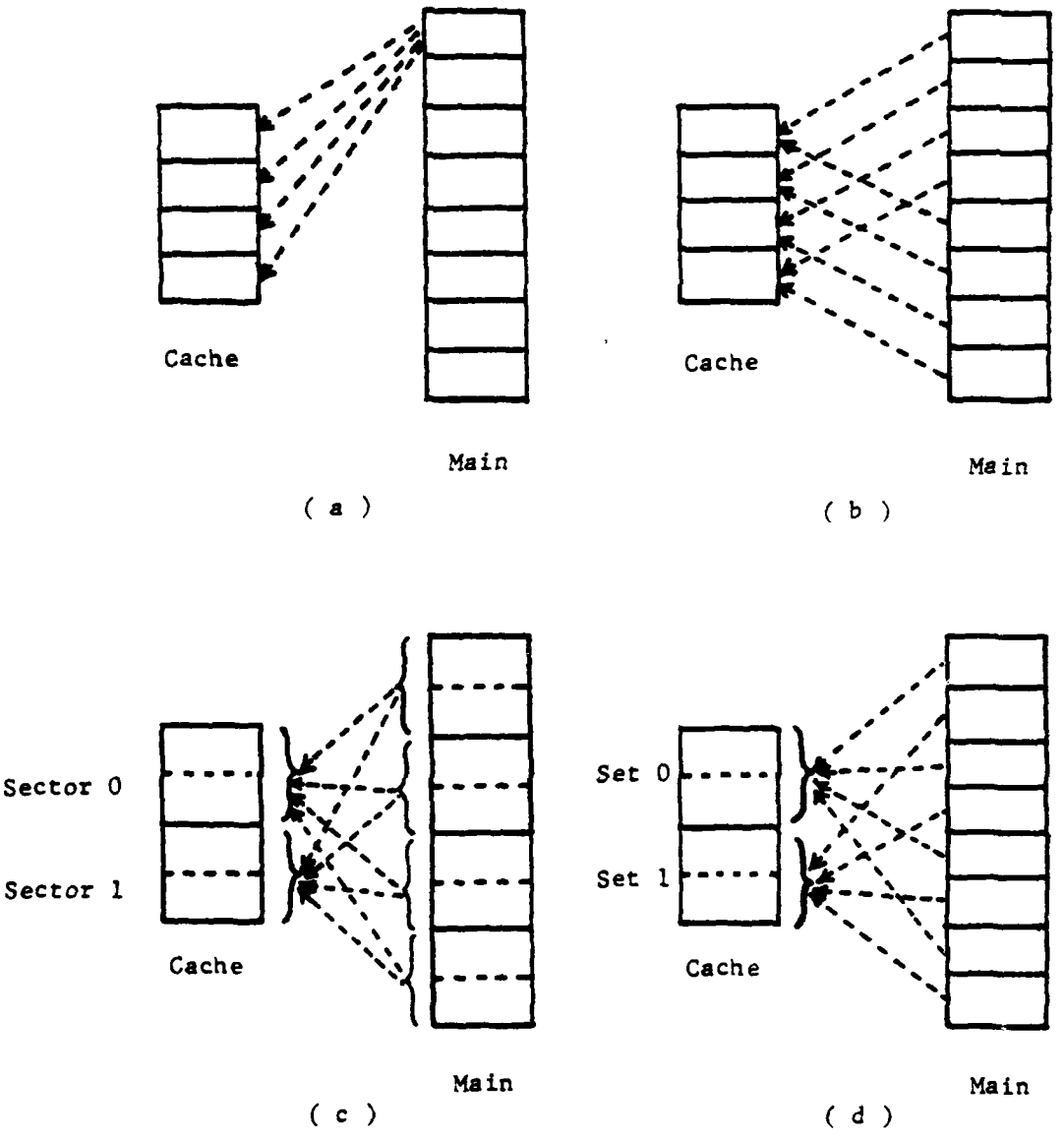


Figure 2.2.1 Cache memory mapping mechanism:
(a) fully associative mapping, (b) direct mapping,
(c) sector mapping, and (d) set associative
mapping.

associative map is used. Obviously, this fully associative lookup is an intolerable bottleneck in the system. A more detailed analysis below shows that a severe bottleneck problem is inevitable if the fully associative scheme is used in a multiple-stream computer system with shared cache memory.

Generally speaking, an associative lookup table (or cache directory) can be implemented explicitly or implicitly. Here an explicit lookup table allows the referenced data to be accessed only if the corresponding tag has already been checked; an implicit lookup table allows the tags and the corresponding data to be accessed simultaneously. For an explicit lookup table, the lookup table is usually physically separated from the cache memory module. However, for an implicit lookup table, block addresses and data are usually stored together in the same cache module. An explicit lookup table is usually built inside the processor and is used when a procesor needs fast interrogation service. An implicit lookup table provides the ability for readout and interrogation to be achieved almost simultaneously. Implicit table implementation in a shared fully associative cache can accept only one request at any time because parallel (or associative) searching through the whole cache memory is required. Therefore, a bottleneck occurs at the shared cache memory in the implicit lookup case. In the explicit lookup case, if each processor is allowed to have its own local lookup table, which contains the information about its own cache usage, then the multicopy problem occurs in the local lookup tables because the cache is shared among all processors; if each local lookup table contains the information about

overall cache usage, then maintaining all local lookup tables is practically infeasible with present technology. As an illustration, consider a parallel-pipelined processor of order (s,p): there are p simultaneous requests issued every STU. At any time, if all p requests result in cache misses, then there may be sufficient time to update p entries in each lookup table because the block transfer time is usually much longer than the lookup table cycle. However, in the case of p-1 simultaneous misses, the lookup table has to be so fast that it can update p-1 entries, i.e. p-1 write cycles for single-ported lookup table, within one STU in order to accept the next request made by the only currently hit process. When the number of processors, p, is large or the segment time unit, τ, is small, to meet this speed requirement for the lookup table is not trivial. If these lookup tables are also used by the replacement algorithm, then the situation becomes even worse since the new state of each local lookup table critically depends on the results of all p simultaneous requests. In order to avoid the multicopy problem in the local lookup tables, the explicit lookup table has to be centralized and shared. However, the lookup table maintenance problem still exists and access conflict may occur at this centralized lookup table. For these reasons we exclude the fully associative cache from further consideration in the case of shared cache memory design.

2.2.2 Direct Mapping

At the other extreme is direct mapping. Cache memory and main memory

are logically divided into equal size blocks. Each block has associated with it its own specific tag. In this scheme, if there are N blocks in cache memory, then every Nth block from main memory may be mapped into one specific block of cache memory as shown in figure 2.2.1(b). The tag associated with each block is actually the high order bits of CPU-generated address. At the time of each processor request, the high order bits of the CPU-generated address are compared to the tag of the cache block to determine whether the requested data is stored in the addressed cache block. Because of the direct mapping aspect, there is one, and only one, block of cache memory which can store a specific block of main memory. Therefore, no associative lookup search is needed. The tag comparison is achieved by using implicit lookup. Not only is the hardware needed to provide direct mapping very simple, but also the cache access time is small because the desired data and the desired tag can be accessed simultaneously. A disadvantage of direct mapping in a uniprocessor environment is that the cache hit ratio drops sharply if two or more blocks, used alternately, happen to map onto the same block in the cache. The possibility of this contention may be small in a uniprocessor system if such blocks are relatively far apart in the CPU-generated address space. The possibility of this contention in a multiple-stream shared-cache system may be much higher than that in a uniprocessor system because many concurrently active streams are sharing the cache. As can be expected, the more streams in a shared cache system, the higher the probability of contention. Thrashing may occur while many streams are contending for a single block in the cache. Here thrashing means that a just-replaced block is needed again immediately

due to a cache miss. This phenomenon of excessively moving blocks back and forth between cache memory and main memory can keep the cache busy and the processors idle most of the time. Deadlock may also happen if the processors are not allowed to access main memory directly and the shared cache does not have the ability of load through [52]. Two processes are deadlocked if neither can continue until the other continues. Load through is simply the ability to by-pass the cache for the specific data referenced when data is not found in the cache. Thus, the data arrives at the CPU as fast as it could from a main memory in a noncache organization. Without load through, a just-brought-in block required by a specific processor may be replaced due to a cache miss by any other processor before the request of this specific processor is satisfied. Due to the high possibility of cache block contention and performance collapse, direct mapping should also be ruled out for a shared cache memory design.

2.2.3 Sector Mapping

In sector mapping, cache memory and main memory are logically divided into sectors each composed of a number of blocks. A sector from main memory can map into any sector in the cache. The requests to main memory, however, are for blocks and if a request is made for a block not in the cache, the sector to which this block belongs is assigned space in the cache but only the block that caused the miss is brought into the cache and the remaining blocks of this sector are marked invalid. Figure

2.2.1(c) illustrates the sector mapping with a two block sector. Although sector mapping needs relatively few tags (one tag per sector in the cache plus invalid bits), the performance of this mechanism is now known to be unsatisfactory since sectors in cache may not have high space utilization. In a shared cache system using sector mapping, the same bottleneck problem as that mentioned in fully associative mapping will result due to the fact that sectors are randomly mapped. It follows that sector mapping is not an acceptable candidate for shared cache memory design.

2.2.4 Set Associative Mapping

In the set associative mapping mechanism, again, cache memory and main memory are divided into blocks, with each block of cache memory having a tag associated with it. The blocks in cache memory are then grouped into sets. The set size is the number of cache blocks contained in each set. If the cache is divided into N sets, then a block i in the main memory is mapped into the set j in the cache satisfying i=j(modulo N). Each set is conceptually controlled by a small associative memory, so that mapping within each set is fully associative. In figure 2.2.1(d), a set associative mapping with a set size of two is shown. Set associative mapping reduces to direct mapping when the set size equals one; it reduces to fully associative when the total number of sets in the cache equals one. Intermediate set sizes lead to mapping methods requiring an intermediate amount of associative hardware. For each set

of size s the associative mapping within the set permits any s blocks, selected from those which belong to this set, to be stored in the cache simultaneously.

It has been shown for uniprocessors that set sizes of 2 or 4 under set associative mapping perform almost as well as fully associative mapping at little cost increase over direct mapping [46]. The set associative mapping mechanism has become widespread for the operation of cache memories for reasons of cost and efficiency. In a multiple-stream shared-cache system using set associative mapping,there exists no parallel lookup bottleneck problem such as that in the fully associative and sector mapping schemes since each set in the shared cache is uniquely addressable by all the processors. Block access contention may still occur within each set in a shared cache memory system. This contention can be reduced to very low levels by choosing a reasonably large set size. The appropriate set sizes for multiprocessors with shared cache will be evaluated later. Each set in the shared cache system can only accept one request per STU for single-ported lookup tables because an associative search is needed for blocks within each set. This conflict problem can also be reduced to very low levels by using an appropriately large number of cache memory modules each of which contains few sets or only one set. Then this conflict problem is essentially the same as the memory conflict problem in multiprocessor systems with shared main memory. Since associative mapping is used only for blocks within each set, each associative lookup table is updated if and only if a request has been made to and accepted by the associated set. Note that there is

at most one such request per STU for each set. Only the results of this request can make the referenced associative lookup table change to a new state. The time needed to update the associative lookup table is transparent. As shall be seen, the set associative mapping mechanism is the most suitable and promissing candidate for shared-cache memory system design if a set of proper component parameters is chosen. The rest of this thesis is based on the set associative mapping mechanism, except where mentioned.

2.3 Management of Cache-Miss

Another important cache memory management decision is the choice of replacement algorithm which determines which block or blocks will be removed from cache memory in order to make space available for new blocks. This choice is constrained by the selected mapping mechanism. For set associative mapping, a cache miss when referencing a particular set causes some block in that set to be replaced. Various page replacement algorithms have been proposed [23,53]. Since it is generally impossible for the cache to keep the working set [24] of even one program in the cache at any time, and also because of the high speed requirement of the cache controller, only simple and fixed space replacement algorithms such as LRU, FIFO, and RAND are generally considered. The LRU (Least Recently Used) replacement algorithm replaces that block in a set which has not been referenced for the longest period of time; the FIFO (First In First Out) replacement algorithm replaces that block in a set

which has been in the cache memory for the longest period of time and the RAND (RANDom) replacement algorithm selects from the set a block to be replaced at random.

2.3.1 Replacement Strategies

FIFO has the advantage that it is easily implemented. Unfortunately, this method has the defect that some frequently used block, e.g. one contained in a program loop, may be replaced because it is the oldest block, yet it may be the block referenced next. FIFO has been shown to exhibit other anomalous behavior for certain reference strings [54]. This page fault anomaly is the phenomenon that increases in memory size can also increase the number of page faults, i.e. FIFO is not a stack algorithm in the sense of [23].

RAND is a very simple and naive procedure. RAND, like FIFO, may also replace a frequently used block because the replaced block is randomly selected. Although RAND yields acceptable performance when set size is large enough, the cache tends to be expensive and/or slow for large set sizes.

The LRU replacement algorithm is based upon the very reasonable and empirically justified assumption that the least recently used block is the one least likely to be referenced in the near future. Thus the LRU algorithm tends to avoid the replacement of frequently used blocks, in contrast to FIFO and RAND. The LRU replacement algorithm has a

characteristic very similar to that of working set replacement policy. Both these algorithms are adapted well to program behavior and based on the principle of locality of reference. The LRU algorithm is a stack algorithm and therefore the hit ratio increases monotonically with the memory size [23]. This characteristic guarantees that the page fault anomaly mentioned above will not occur with LRU.

Comparisons between RAND, FIFO and LRU were made by Belady [53], and it was observed that RAND and FIFO gave similar performance results. LRU gave improved results. Hence only the LRU replacement algorithm will be considered further in this thesis. In the multiple-stream processor system with set-associative shared-cache using LRU, each set has its own hardware-implemented LRU replacement algorithm. All the streams in the system thus compete for cache space by competing for their share of the blocks within each set.

A very important criterion for the LRU replacement hardware design is speed. The set size chosen should represent a compromise between minimizing the miss ratio and minimizing the speed and cost. Small set sizes result in competition among simultaneously active blocks for the small number of spaces in a set; large set sizes require more hardware for search and replacement and/or operate more slowly. The choice of set sizes and other component parameters will be discussed when the simulation results are analyzed in chapter 4.

2.3.2 Cache Block Fetch and Handling of Write-Miss

So far the fetch strategy, i.e. demand or prefetch, has not been resolved. The fetch strategy defines the policy of when to load blocks and how many blocks to load at a time. Since cache size is much smaller than main memory size, the space contention in a cache is more severe than that in the main memory of a paging system. Block prefetch is normally not used at the cache level. Usually the cache is so small that it cannot hold the entire working set of a program. Thus, it is very possible that some frequently used blocks would be expelled in order to allocate prefetched blocks. The prefetched blocks may not be referenced in the near future or may not be referenced at all due to changes of program locality. All the previous studies, i.e. [45,46-48,51], of cache design show that the hit ratio drops if the block size exceeds certain minimal values. This well-known result illustrates that sequential block prefetch may not be suitable for cache design. In this thesis, only demand fetching, i.e. fetch one block on miss only, is considered.

There are two possible variations of demand fetching, that is, write allocation and no-write allocation [45,46]. No-write allocation brings a block into cache only on a read miss; write allocation brings a block into cache on read as well as write miss. Since these two fetching strategies are tightly related to the manner in which processor writes to memories are handled, the determination of fetching strategy will be deferred until the main memory updating scheme is chosen.

Basically, there are two kinds of main memory updating schemes. In the simplest approach, known as write through [45,55,56], all writes are sent to main memory, the block is also updated in cache if it is there. In this case, it is implied that the processor has the ability to directly access the main memory and the blocks are brought into cache only on misses for processor read-memory requests. Hence write through is usually used with no-write allocation. Write through is also called store through by some authors [17,52]. An alternative approach, known as swapping [52], is to write the word in the cache and then always write the word in main memory when the block in cache must be replaced. Since direct modifications of data always happen in the cache, this approach is usually used with write allocation. Swapping is also termed write back [45], block swapping [46], conflict-use-writeback [55], or nonstore-through [17].

The effectiveness of write through is known to be less than that of swapping in a uniprocessor system [52,46,55]. A lot of unnecessary writes may occur with the write through strategy because a word may be updated several times during the time it resides in the cache. The statistics of instruction mixes show that, depending on the processor architecture, from one tenth to one third of all accesses are write accesses [17,55]. The access rate to main memory cannot be less than the write access rate of the processor if a write through updating strategy is used. Recall that in order to gain maximum advantage from the cache, the fraction of references to main memory must be minimized. This points out that pure write through is not suited to high performance systems.

Although write through can be overlapped with cache cycles because the write operation has no deadline, the processor will be forced to wait on the successful completion of writes to main memory if there is no buffer used for writing the main memory. Recently, Smith [56] reported that a sufficiently large buffer for write-through can greatly reduce the performance difference between write-through without buffering and swapping.

On the other hand, in addition to conceptual simplicity and ease of implementation, write-through has the advantage that obsolete information is never present in main memory. This can be particularly valuable in systems with independent input/output paths. Note that the input/output channels have data rates substantially lower than instruction processors and therefore there is no performance advantage in connecting them to a high speed cache.

The advantage of the swapping schemes is that, at least in theory, the access rate to the main memory is the miss rate and may be reduced to any desired value by a sufficient increase of the cache size. However, the swapping schemes have the disadvantage of invalid multiple, nonidentical copies of the same data existing in several memories. Pohm [57] identifies three swap algorithms: simple swap, where information is always written back when it is removed from the cache; flagged swap, by which only information that has been changed is written back to main memory; and flagged register swap, by which a block of information in the cache that has been changed and is to be written back is swapped into a register buffer and then written to the main memory after the fetch has

been completed. It is shown [57] that a flagged register swap algorithm yields the best performance among all the above three schemes and the write-through scheme without buffering.

Up to now, discussion about main memory updating schemes has been based on a uniprocesor system or a system with a single CPU and several asynchronous input/output channels. Unfortunately, the above results and conclusions cannot be directly applied to multiprocessor systems. The reasons are explained below.

As shown in figure 1.5.1, there exists an interconnection network between the caches and the main memory modules in a private-cache multiprocessor system. If a write through scheme is used in such a system, the high write access rate to the main memory will increase the main memory conflict. Note that in a multiprocessor system with p streams and p private cache memories, the write access rate to the main memory is p times higher than that for a uniprocessor system if both systems use write through. In addition to the access conflicts between the block transfer operations, access conflicts between write accesses will take place as well as interference between block transfer operations and write accesses. Furthermore, the interconnection network is switched very often due to the high write access rate. This will cause setup time overhead if a slow switching network is used. The main difficulty is that write through will cause the multicopy of shared data among private cache modules. In section 1.6, two previously proposed solutions [16,17] of the multicopy of shared data problem have been discussed. In these two algorithms, swapping is implied because both algorithms treat a block

as a critical section when this block is being modified. All the modifications of data have to be done in all the relevant caches and then the blocks are copied back to the main memory when replaced. The swapping scheme with write allocation is the primary requirement for these algorithms. Consider that if write through is used, then every single word write access becomes a critical section in order to solve the multicopy of shared data problem. Clearly, this is an intolerable design and swapping with write allocation is preferred in a private-cache multiprocessor computer system.

Due to the fact that there is no switching network needed between the cache and the main memory in a shared-cache multiple-stream system (see section 2.7 for explanation), write through in such a system will not cause severe main memory access conflict. Updating the main memory can be overlapped with the processor cycle if sufficiently large buffering for write through is provided. In this case, the interference between write accesses and block transfer operations on the same bus line can be supressed if a higher priority among main memory accesses is granted for block fetches and lower priority for queued write accesses to the main memory. It might happen that a read access to a block in the main memory occurs due to cache miss before this block is updated to reflect pending writes. This situation can easily be handled by adding some hardware control in the main memory. Every block read access to the main memory then searches the corresponding buffer to determine whether the referenced block needs to be updated. This buffer search time can be expected to be small compared to the total block transfer time and can be

included in the block transfer time, T, for analytical purposes. The write-through with buffering scheme discussed above preserves the advantages and eliminates the disadvantages of a simple write-through scheme. It may also enhance the hit ratio since all the write accesses are considered as hit requests. Therefore, the write-through with buffering updating scheme and the no-write allocation strategy are desirable in a shared-cache multiple-stream computer system.

To be able to make a fair comparison between private-cache and shared-cache organizations, a flagged register swap algorithm is considered in the private-cache multiprocessor systems. Then, updating the main memory is overlapped with processor cycles in both systems. Similarly, for private-cache systems, the register search time can be considered as part of the total block transfer time. Note that the major interest in this thesis is the performance variation due to the memory conflict problem and the hit ratio behavior for a range of cache organizations. The effectiveness of particular buffer sizes for various main memory updating schemes is not investigated. In the rest of this thesis, it is assumed that a sufficiently large buffer size is used in all models such that the processors will never be blocked due to the main memory updating operations caused by either the write through or swapping policy.

The cache management options presented so far, such as mapping mechanisms, replacement algorithms, and write policies (write through and write back) do not affect the analysis presented in chapter 3.

2.4 L-M Cache Configuration

In the previous sections, several functional parameters of cache memory design have been discussed. Before any analytic model can be developed for shared-cache memories, a shared-cache memory organization has to be chosen in order to determine cache memory request scheduling. One memory organization which is suitable for multiple-stream processor architecture is reviewed in this section.

The memory organizations for multiprocessor systems discussed by most authors use N address busses for N independent memory modules. Although the performance of such a memory organization is good, the bus cost and the interconnection network cost are usually high. Recently, a two-dimensional memory organization which intends to increase the bus utilization and reduce the cost of both the interconnection network and bussing is investigated by Briggs and Davidson [41]. This organization, referred to as the L-M memory organization, can achieve high performance without sacrificing cost-effectiveness and is used as the shared cache memory organization in this thesis. The rest of this section reviews the L-M memory organization and explains the way that this organization is adapted for shared cache memory applications.

Many large scale integrated RAM chips have their address cycle significantly smaller than the memory cycle. Therefore, more than one module can share one address bus thereby increasing the bus utilization and reducing the bus cost. To allow maximum address bus sharing, the address hold time, a, on the address bus should be designed as short as

possible. This may be achieved by incorporating an address latch within each memory module. Then, the address hold time on the bus for each memory request is the address bus cycle time which is only a small fraction of the memory cycle time. Whenever a memory module is active for a memory cycle, its associated address bus is active for only this small fraction of the memory cycle. By multiplexing a group of memory modules on an address bus, the period for which the bus is inactive may be used to broadcast the addresses of new memory requests. Although the performance is degraded as a consequence of bus-sharing, some address busses can be eliminated and the size of the processor-memory interconnection network can also be reduced. This tradeoff between bus-sharing and performance will be studied in later chapters. Similarly, data busses can also be time-multiplexed if latches are provided within each memory module to gate-in write data and drivers can be enabled to drive out read data. For simplicity, it is assumed that the address bus is busy as long as there is some cache module on that bus which is involved in a block transfer operation. Hence the data busses do not pose a limiting constraint and are not explicitly considered from now on.

In the following discussion, a <u>line</u> is used to denote an address bus within the memory. Hence, assuming that there are N identical memory modules in the shared cache memory, there can be up to N independent lines in the memory. The L-M memory organization, shown in figure 2.4.1, consists of ℓ ($=2^k$) lines and m ($=2^{n-k}$) memory modules per line, such that a total of N ($=2^n = \ell m$) identical memory modules are arranged in a

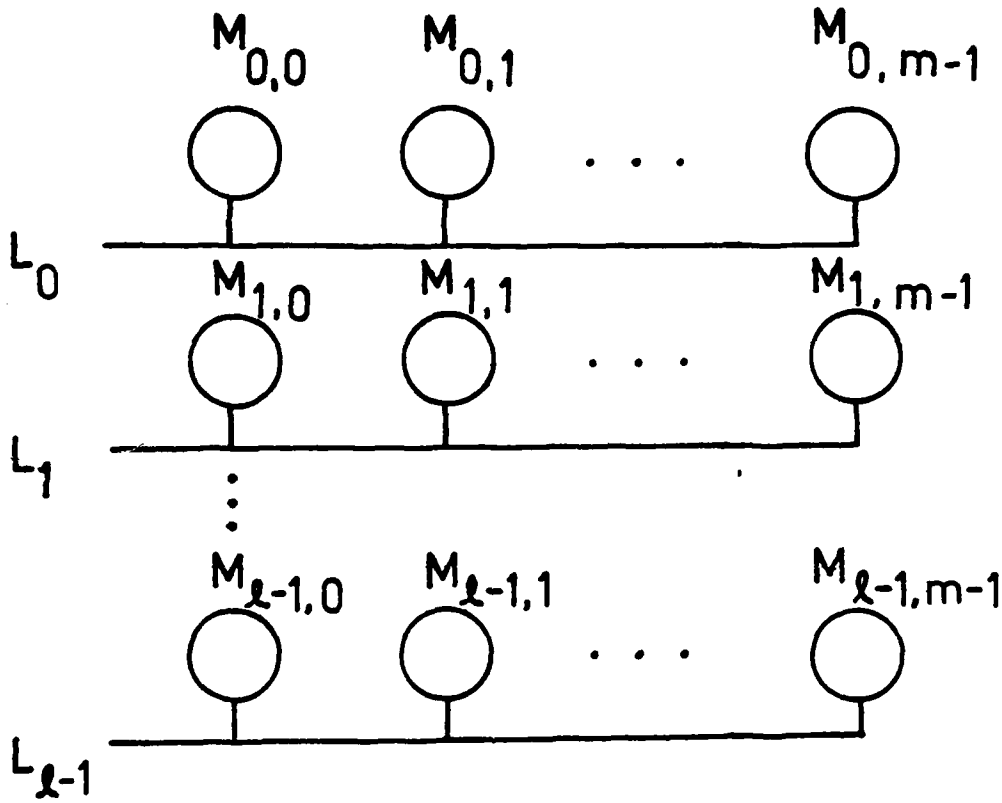


Figure 2.4.1 L-M memory organization.

two-dimensional matrix form, where both k and n are integers and $0 \leq k \leq n$. Therefore, a particular configuration of the L-M memory organization can be characterized by the corresponding memory configuration, (ℓ,m). In figure 2.4.1, L_i and $M_{i,j}$ represent line i and module j on line i, respectively. It is obvious that no line sharing is required if the segment time is greater than or equal to the memory cycle time (c=1). In this case, the performance is independent of the number of memory modules per line. Figure 2.4.2 shows the bus structure of the L-M memory organization. Each set of modules on a line in addition to sharing the same address bus share the same data input and data output busses. That is, there is one each of address, data input and data output busses for the set of m cache memory modules on each line.

2.5 Address Interleaving

Memory interleaving is a common and inexpensive way to yield high effective memory bandwidth. In a multimemory system with N memory modules, if successive word addresses are assigned across the memory modules, modulo N, the memory is interleaved by low-order bits, or interleaved by words. The low-order log N bits of each address indicate the module number in this case. However, this low-order-bit interleaved memory is not suitable for shared cache memory applications for the following reasons :

(1) While executing the block transfer operation due to a cache miss,

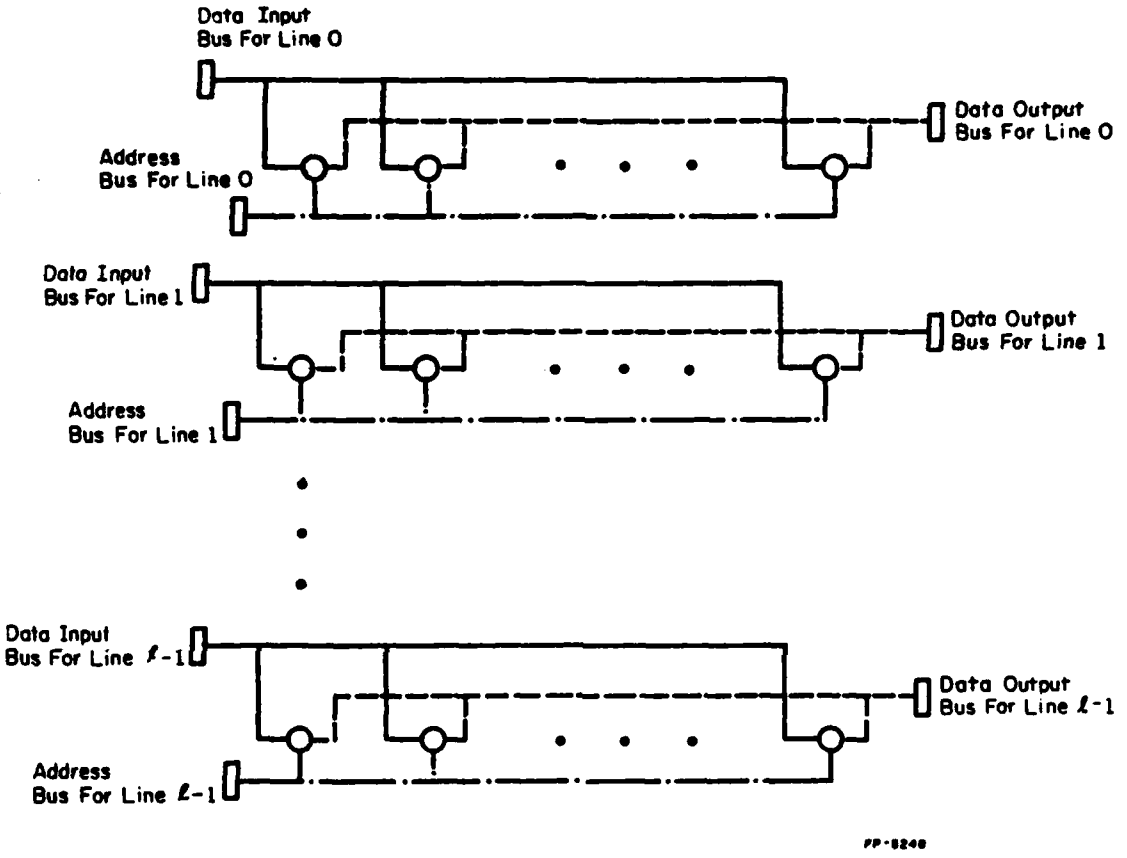


Figure 2.4.2 Bus structures of the L-M memory organization.

the whole cache system will be busy and all the processors will be blocked for the block transfer period. One stream with poor cache performance may then degrade the performance of all the other streams in the system.

(2) For the set associative cache memory mapping mechanism, each block of cache storage has an identifying tag associated with it. If low-order-bit interleaving is used in the shared cache memory, then the words in one block will be spread over several, or perhaps all, cache memory modules. So, the number of tags required may be up to N times the number of blocks contained in the cache memory if an implicit lookup table is used. In this case, the cache memory cost is high because many tags are needed. In the extreme, when the block size is smaller than the number of cache memory modules, then each word in the cache memory needs its own associated tag. Unfortunately, an explicit centralized lookup table cannot be used because it would cause the same bottleneck problem as that discussed in section 2.2 for the fully associative mapping mechanism. Therefore, a costly cache memory is inevitable if the shared cache memory is interleaved by words.

On the other hand an uninterleaved memory, in which the high order log N bits of each address determine the memory module, cannot effectively reduce memory access conflict in multiple-processor environments.

Therefore, in order to keep the number of tags equal to the number of blocks in the cache memory and to try to enhance the effective cache memory bandwidth at the same time, a compromise solution of address interleaving is proposed here. A memory is called interleaved by sets if the successive set numbers are assigned across the cache memory lines. Figure 2.5.1 shows the address format for a set associative cache memory organization, the least significant b bits of the address determine the word address within a block, the next higher order d bits of the address determine the set address in the cache, and the remaining (high order) bits are used as the tag to identify the particular block within the set. Integers b and d specify the block size ($=2^b$ words) and the total number of sets ($=2^d$ sets) in the cache memory, respectively. The right-most b bits of the address are then referred to as word bits; and the next d bits of the address are called set bits. The shared cache memory can easily be interleaved by sets by choosing either of the following two possible implementations.

The address format for the first implementation is shown in figure 2.5.2(a). Assume that the total number of sets is greater than or equal to the total number of cache modules. Given the memory address of a word, the k least significant bits of the set bits address one of the 2^k lines, L_i, and the next higher order n-k bits address one of the 2^{n-k} modules on line i, $M_{i,j}$. Then the high order d-n bits of set bits address one of the 2^{d-n} sets in module $M_{i,j}$ and the tag bits are used for associative search. If the associative search results in a hit, then the b block bits address one of the 2^b words in the hit block; otherwise, a

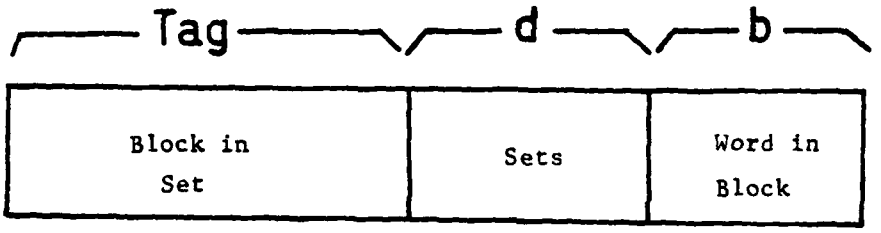
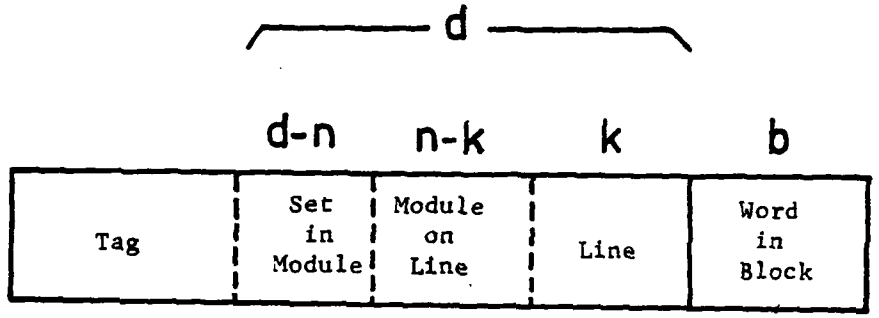
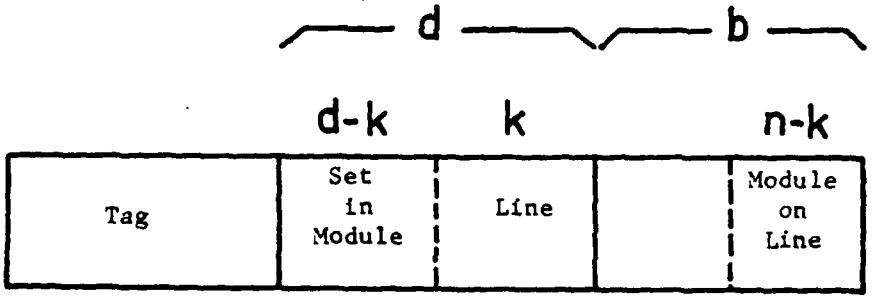


Figure 2.5.1 Address format for a set associative cache memory organization.





$$\ell = 2^k$$

$$m = 2^{n-k}$$

$$N = 2^n = \ell \cdot m$$

Figure 2.5.2 Address formats for two implementations of interleaving by sets.

miss occurs and a block transfer operation may be initiated according to some fetch policy, discussed in chapter 3. Hence, the cache modules are interleaved on the low order n bits of the set bits and the lines on the low order k bits of the set bits. Note that the set size is not explicitly indicated in the address format and is dependent on the block size, the total number of sets and the total cache capacity. Although this implementation requires no tag duplication because each set is wholly contained in a particular cache module, potential serious performance degradation may result from blocking due to the long block transfer time required. This implementation requires new blocks to be loaded into the cache one word per cache cycle during the block transfer operation. Hence the block transfer time, T, is primarily dominated by the cache cycle time, c, and the block size if high main memory bandwidth is provided. As mentioned before, while a module is involved in a block transfer operation, the associated line is busy until the block transfer operation is completed. Therefore, for a slow cache or a large block, the block transfer time is long and all the cache modules on a line being used for block transfer will be blocked for a long period.

Another implementation tries to improve the performance by reducing the block transfer time without changing the system organization. Assume that the block size is greater than or equal to the number of cache modules per line and the number of sets is greater than or equal to the number of lines. Figure 2.5.2(b) illustrates the address format for this implementation. Given the address of a word, the k least significant bits of the set bits address one of the 2^k lines, L_i, the n-k least

significant bits of the word bits address one of the 2^{n-k} modules on line i, $M_{i,j}$, and the high order d-k bits of the set bits address one of the 2^{d-k} sets in module $M_{i,j}$. Since the least significant n-k bits of the address, the word bits, determine the module number on some line, the successive words in a block are interleaved across the modules on the same line. Hence the modules on one line can be considered as a memory interleaved by words. During the block transfer operation, those modules on the same line can not be cycled synchronously because the associated line is time-multiplexed. However, they perform like phased memories (or interlaced memories) in which each of the memory modules is cycled on a different minor clock cycle. The minor clock cycle here is the signal propagation time from main memory to cache memory, usually made equal to the bus cycle time (1 STU) discussed above. Therefore, new blocks are loaded into the cache one word per bus cycle, instead of one per cache cycle, during the block transfer operation. By properly choosing the number of modules per line, a phased memory system can have the same bandwidth as that of a parallel accessible memory system with the same cycle time and the same number of modules. Note that the L-M memory organization is actually a parallel phased (or interlaced) memory organization. Note that successive sets are assigned across modules on distinct lines and successive words of a block are assigned across the modules on the same line. Although this implementation could reduce the block transfer time, more tags may be needed due to the fact that each block is spread over m cache modules. For a cache with an implicit lookup table, one tag is needed for each block within every cache module. Therefore, over the entire cache each tag is replicated m times.

However,if an explicit lookup table is associated with each line (as discussed in section 2.7), then no extra tags are required. Such an explicit lookup table can be used without degrading performance if the lookup table cycle time is less than or equal to one STU.

These two possible implementations become the same if m equals 1, that is one module per line. The choice between the two implementations involves the tradeoffs of performance and cost, i.e. block transfer time and number of tags.

For brevity, a shared cache memory interleaved by sets with a memory configuration characterized by (ℓ,m), is a particular realization of the L-M memory organization, where the number of lines, $\ell = 2^k$, and the number of modules per line, $m=2^{n-k}$. For example, if k=3, n=5, then ℓ=8 and m=4. Hence we have a memory configuration of $(\ell,m)=(8,4)$. The total number of sets in the cache and the block size are determined by d and b, respectively. For example, given a 4K cache memory with b=4 and d=5, then block size is 16, number of sets is 32, number of blocks is 4096/16 = 256, and set size is 256/32 = 8.

2.6 Shared Cache Request Scheduling

In section 2.4, it was assumed that address and data input latches and selectable data output drivers exist in every cache memory module to allow minimum bus usage time for each request in order to achieve maximum line sharing. It has been shown in a previous study [41] that

performance decreases as a, the address hold time on the bus (or address bus cycle time), increases. Since a=1 is realistic, simpler to design with and model and allows fewer lines ($\ell > ap$ is recommended by this previous work), the address bus cycle time is then assumed to be less than or equal to one STU throughout this thesis, i.e. the address hold time on the bus, a=1.

Recall that a parallel-pipelined processor of order (s,p) issues p simultaneous cache requests each STU. Of those p parallel requests, some of them might address the same line resulting in a conflict. Even when all p simultaneous requests address distinct lines, conflict can still result if a request addresses a line which is still executing the block transfer operation for a previous cache miss or a module which is still executing the cache cycle for a previous cache request. Such a line or module which is serving a prior request at time t is said to be busy or active at time t. If a line or module is not busy, it is said to be idle or inactive.

Definition 2.6.1 A cache memory request collision is said to occur when a cache memory request attempts to access a busy line or module, or when at least two simultaneous cache memory requests attempt to access the same line. □

When more than one request attempts to access the same line simultaneously, a multiple access line collision occurs. When a request passes through the multiple access line collision and attempts to access

a busy line, a line collision occurs. Similarly, a module collision occurs when a request passes through both the multiple acccess line collision and the line collision and attempts to access a busy module.

Definition 2.6.2 The status of module $M_{i,j}$ at time t is L_i busy or idle at time t, and $M_{i,j}$ busy or idle at time t. □

Both the status and the content of a memory module addressed by a request are required to determine the outcome of the request. A request can access module $M_{i,j}$ at t if and only if $M_{i,j}$ and its line L_i, are both idle at t. Hence a request is rejected if it addresses a busy line or module. However, if all simultaneous requests to a line refer to idle modules on that line, one of these is accepted and the others rejected. A request is termed an acceptable request if it addresses an idle module on an idle line. If there is only one such request for a line, the request will be accepted. When there is more than one simultaneous request for an idle line, one of them is selected arbitrarily and the others are rejected. The selected request is then accepted if and only if it addresses an idle module, i.e. if and only if it is acceptable.

Definition 2.6.3 A request is either successful or unsucessful (blocked). A successful request is an accepted request which results in a hit; an unsuccessful request is either a rejected request or an accepted request which results in a miss. □

A request goes through the memory system in the following way.

Suppose it gets initiated on line L_i at time t. Then it keeps that line busy for one time unit in the interval (t,t+1). Following this, it initiates a memory cycle on the addressed module $M_{i,j}$, which keeps that module busy for c time units in the interval (t+1,t+c+1). In other words, the line L_i is busy at time t, and the module $M_{i,j}$ is busy with respect to other requests at times t+1, t+2,..., t+c-1. However, following an initiation of a block transfer at time t on line L_i and module $M_{i,j}$, both L_i and $M_{i,j}$ are busy with respect to other requests at time t, t+1,..., t+T-1 if the implementation of figure 2.5.2(a) is used. Hence L_i and $M_{i,j}$ remain busy in the interval (t,t+T). If the implementation of figure 2.5.2(b) is used, then all the modules on the same line will be involved in the block transfer operation when that line is used for block transfer.

One simple method of handling the unsuccessful requests is to recycle their processes through the processor segments and resubmit them as new cache memory requests one instruction cycle later. During the recycling of each unsuccessful request, a control flag is set for the process which originated the unsuccessful request to deactivate execution of that process until the request is satisfied (successful). Once the request is satisfied in some later cycle, the flag is reset and execution of that process is reactivated. This method for suspending process execution requires no distinction between an unsuccessful request due to conflict rejection and cache miss. Actually, from the procesor point of view, the cache memory is a resource with constant service time. Therefore, the processor should receive the data for a read access or the

completion signal for a write access by a certain deadline after a request has been made. Otherwise, the request is unsuccessful and the control flag is set automatically.

As discussed in section 2.2.2, deadlock might happen due to the space contention in a system with small set sizes and a large number of streams. A modified LRU replacement mechanism can eliminate this deadlock possibility. Here the modified LRU replacement mechanism is a simple LRU with the added property that once a block is brought into the cache it does not become eligible for replacement until it has been referenced at least once. For simplicity, whenever a request results in a cache miss, this miss request will be rejected if the least recently used block in the set has not been referenced at least once. In this case, no block transfer operation will be initiated and the modified LRU will not change state. This rejected miss request will be resubmitted next instruction cycle. Otherwise, a normal LRU replacement mechanism is used.

Another possible method of handling the unsuccessful requests is that processors do not resubmit the unsuccessful requests due to cache misses and resubmit the requests only for those requests rejected due to access conflict. Whenever a cache miss occurs for a request made by one particular processor, this processor is held idle until the data which caused the miss is loaded from main memory directly to the processor. During the waiting period, this process simply makes null passes and makes no request.

In chapter 3, the first method of handling the unsuccessful requests is assumed for developing the basic analytical models. However, the analytical models for the second method of handling the unsuccessful requests can be obtained by direct extension of those basic models with some additional assumptions. This extension is discussed in chapter 4.

In summary, there are basically four different reasons for a request to be blocked. Let h denote the cache hit ratio. A cache memory request may be unsuccessful due to

(1) Multiple access line collision, which may occur only if $p>1$,

(2) Busy line collision, which may occur only if $h<1$ and $T>0$,

(3) Module collision due to a busy module on an idle line, which may occur if $c>1$, and

(4) Cache miss, which may occur if $h<1$.

2.7 System Configurations

Since the cache memory is shared by all the processors in the system, an interconnection network is required to connect the processors with the shared cache memory modules. A p-by-ℓ crossbar is assumed for this interconnection network to simplify further discussion. Although the crossbar can route the accepted requests to the appropriate lines and modules, some functional units are required to accept or reject incoming requests. Basically, the functional units are needed to resolve

conflicts and maintain the status of currently busy lines and modules. There exist a wide variety of possible implementations of such functional units and a particular choice depends on the design objectives. For instance, the functional unit required to resolve multiple access line conflict can be implemented based on a processor priority scheme or a round-robin assignment; and the functional unit required to store and update busy line or module status may be readily implemented by a set of shift registers or a proper amount of content addressable memory. Details of these implementations can be found in [41] and will not be repeated here.

Figure 2.7.1 shows the system configuration proposed in this thesis. Note that there is no interconnection network required between the shared cache memory and the main memory. The main memory is automatically shared by all processors because the cache memory is shared. In other words, all the information which is mapped into the cache modules on one line could be stored in a set of main memory modules associated only with that particular line. Whenever a cache miss occurs for some module on a particular line, the new block to be fetched will be found in one of the main memory modules associated with the same line. Therefore, a single bus is sufficient to connect the cache modules with their associated main memory modules. Note that more than one main memory module interleaved by low-order bits could be attached on each line to provide high main memory bandwidth. The bus width at the main memory and the degree of interleaving of main memory on each line are determined to match the required main memory bandwidth which then defines a specified block

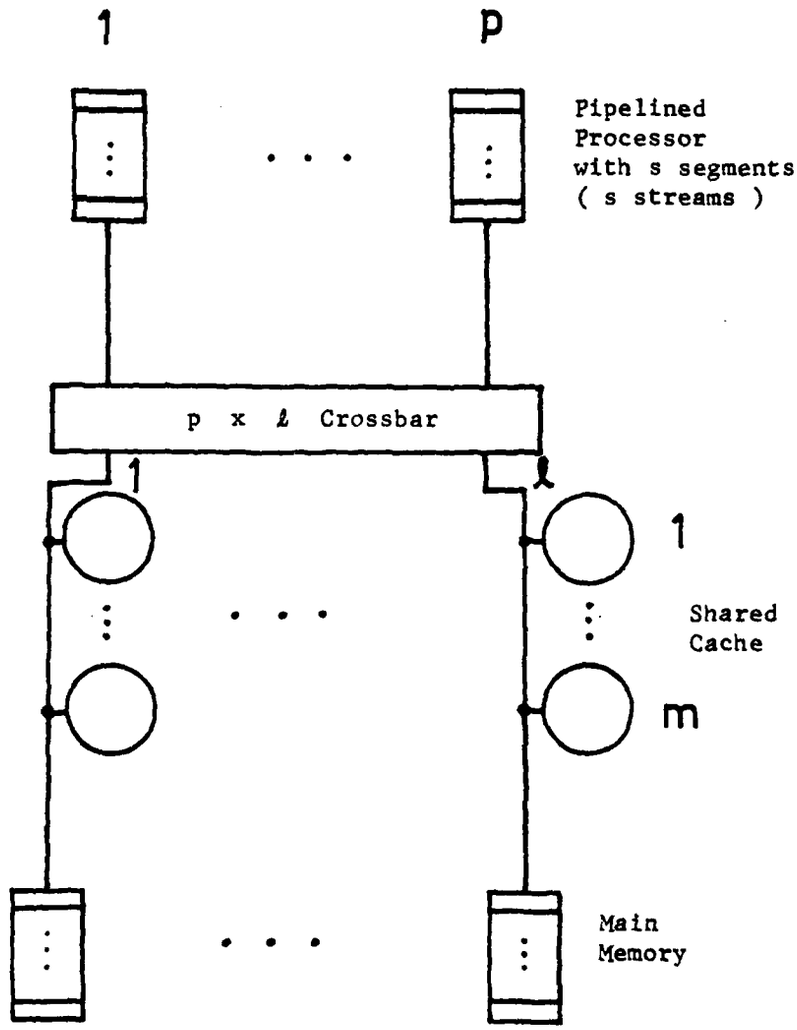


Figure 2.7.1 Shared cache system organization.

transfer time, T. Therefore, only the block transfer time will be considered as a design parameter. The specific main memory design implementation, such as the main memory bus width and the number of main memory modules per line, required to achieve T will not be explicitly considered in further discussion. Since the research focuses on the cache memory problem, the effects of lower levels of the memory hierarchy involved with paging faults, etc., are not included here. Note also that the buffers for write through are not shown in figure 2.7.1.

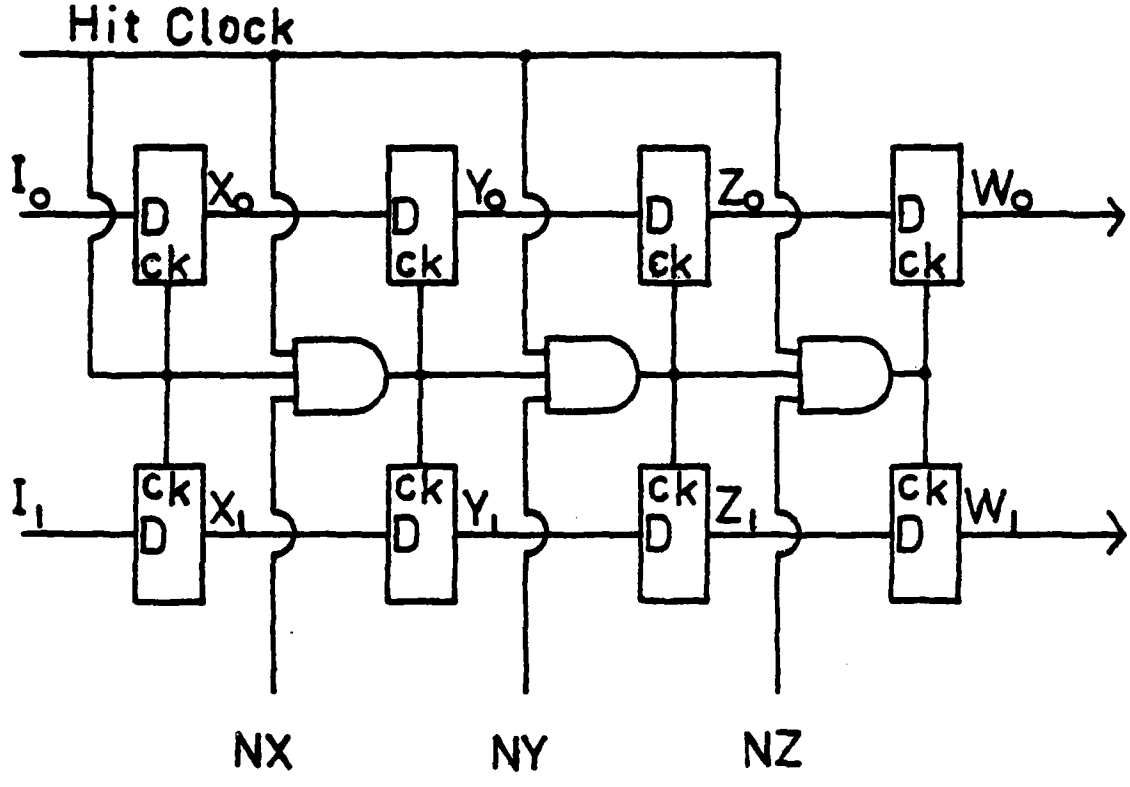
One might be concerned that the crossbar between the processors and the shared cache memory requires longer time to access the shared cache memory than the time needed for processors to access a private cache memory. The crossbar indeed delays the turnaround time for every successful cache memory request. However, the total system throughput for a shared cache memory would not be degraded due to the crossbar if a parallel-pipelined processor, which allows for this larger turnaround, can be designed to provide the instruction execution rate which is equivalent to that of a multiprocessor system with private cache memories. It is clear that the time needed to route a request from a processor through the crossbar to the referenced shared cache module is overlapped with the pipelined processor cycle. The delay due to the crossbar will not play any important role in the system performance as long as the address hold time on the bus, a, is one. In this thesis, a is assumed to be 1 and we assume no performance degradation due to crossbar delay in a multiple-stream shared-cache system.

Recall that there is LRU replacement hardware namely an LRU stack,

associated with each set in the shared cache memory. An important concern in the LRU stack design is speed. Since the time required for the LRU updating process is dependent on the set size, it is desired to design an LRU stack such that the speed is fast enough for the selected set size.

Again, many possible implementations of such LRU stacks may exist but only two apparent examples are shown below. Since there is plenty of time to update the LRU stacks for cache misses, only the updating for hit requests is considered in these examples. The first scheme employs a set of fast counters. LRU is implemented by associating a hardware counter, called an age register, with every block in a set. Whenever a block is referenced, its age register is set to a predetermined positive number. At fixed intervals of time, the age registers of all the blocks in each set are decremented by a fixed amount. The least recently used block at any time is the one whose age register contains the smallest number. This smallest number can be obtained by an associative search of the counters. A special select circuit might be required if there is the possibility that more than one age register has the same smallest number.

An alternative implementation employs a set of D flip-flops to maintain the status of the blocks which currently reside in the cache and a few logic gates can achieve the updating function. For a set containing R blocks, that is for a set of size R, log R bits will be sufficient to address any given block in the set and a total of R log R bits are enough to keep all the necessary information for LRU replacement operation. Figure 2.7.2 shows one example of a LRU stack with set size



$$NX = (X_1 \oplus I_1) \vee (X_0 \oplus I_0)$$
$$NY = (Y_1 \oplus I_1) \vee (Y_0 \oplus I_0)$$
$$NZ = (Z_1 \oplus I_1) \vee (Z_0 \oplus I_0)$$

Figure 2.7.2 An implementation of the LRU algorithm.

equal to 4. In this example, the four words of the stack are denoted as X, Y, Z, and W. Register X corresponds to the top of the stack and the register W the bottom of the stack. Register X contains the block number X_1X_0, register Y contains the block number Y_1Y_0, and so on. The number of the block just accessed is available on lines I_1 and I_0 and the number of the least-recently-used block is available as W_1W_0. There are three control signals, that is, NX, NY, and NZ, each of which controls its corresponding block in the LRU stack. NX is 1 if the just accessed block is not block number X; otherwise, NX is 0. NY and NZ are similar. Whenever a request results in a hit, a hit clock is generated immediately to control the updating process. Each of these three control signals together with the hit clock determine if the corresponding block should be shifted to the right in the LRU stack. The number of the just accessed block is loaded into the leftmost pair of D flip-flops every time a hit in this set occurs. The contents of the other pairs is shifted to the right until the previous position of the just accessed block is reached. The rightmost pair of D flip-flops always indicates the number of least-recently-used block in the set associated with this LRU stack.

Practically, cache memory control can be implemented by using various memory devices such as RAM, CAM, or a combination of both. The lookup tables for a set associative cache organization can be achieved in two different forms, that is, explicit lookup table or implicit lookup table. A set associative cache with an implicit lookup table can be easily implemented by using RAM with a word width of R tags, where R is

the set size. Figure 2.7.3 illustrates this implementation with R equal to 2. A cache word contains word i from block 1 and word i from block 2 of the given set. The tags for both block 1 and block 2 can be stored together with the data in the same cache word. However, in this case, the tag is repeated for each data word. This tag repetition can be avoided if these two tags are stored in some other cache word which can be read out simultaneously with the data. Thus, a fetch from the cache will pull out the two data words in which the requested data could reside plus the two tags associated with that data. The tag from a request's effective address is simultaneously compared to both of the tags read from the cache directory. The result of that comparison will result in gating word i from block 1 or word i from block 2, or neither in case the data is not in the cache. Although some comparators and a multiplexer are needed, no expensive CAM chip is required. The comparison itself can be slow since the result is not needed until the cache cycle is complete. Note also that whether an accepted request results in a hit or miss is determined after the cache cycle has been completed.

For a set associative cache with explicit lookup tables, a combination of RAM and CAM devices can be used. The lookup tables are usually implemented by CAM because of the fast parallel associative search required; the cache modules themselves can be implemented by RAM. Since p parallel pipelined processors issue p simultaneous requests each STU, a centralized explicit lookup table shared by all processors will degrade the system performance drastically due to the intensive access conflicts at the lookup table. Hence the explicit lookup tables have to

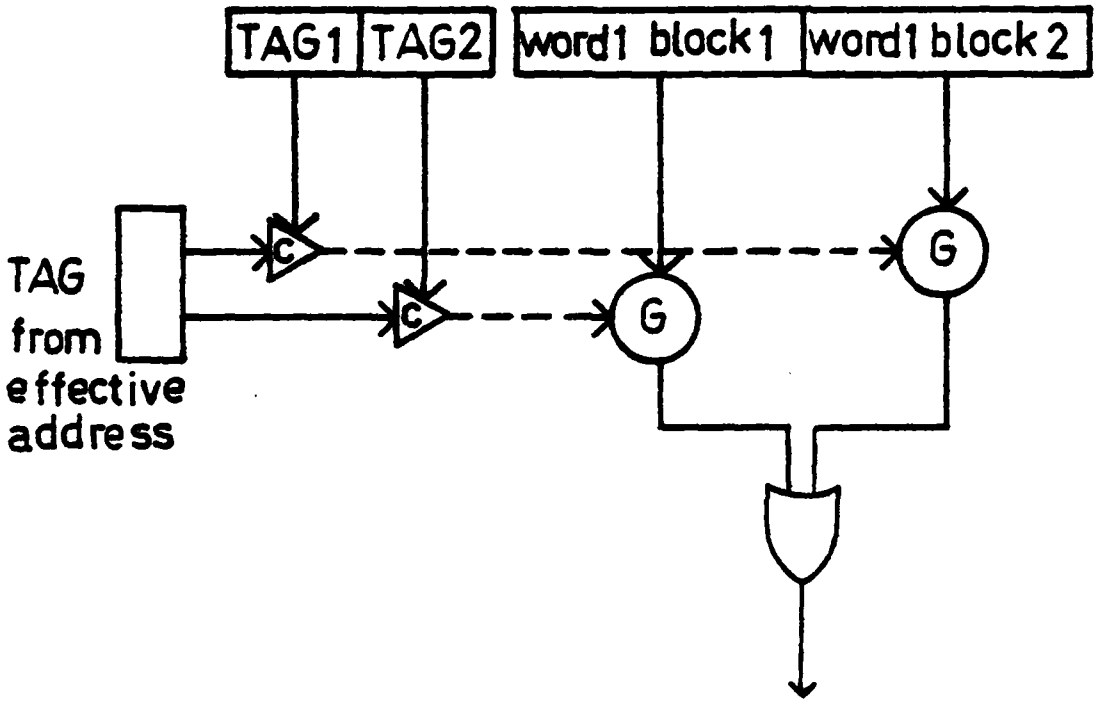


Figure 2.7.3 An implicit lookup table implemented by CAM chips.

be decentralized. One part of the lookup table can be associated with each line or each module as shown in figures 2.7.4(a) and (b), respectively. Since very little time is needed to process a parallel associative search in a CAM, it is assumed that the cycle time of a CAM lookup table is transparent, i.e. it is included in the address hold time on the bus. Then, from the performance point of view, the designs of figures 2.7.4(a) and (b) are equivalent. The implementation of figure 2.7.4(b) will give higher performance than that obtained from the implementation of figure 2.7.4(a) if the lookup table cycle time is greater than one STU. This difference is due to the fact that the table on each line shown in figure 2.7.4(a) poses a limitation on the acceptance of requests: each line can accept at most one request every lookup table cycle instead of one every STU. For simplicity, the time interval needed for checking whether a request is a hit or miss is assumed to be transparent. This assumption can be applied to both the cycle time of a CAM lookup table or the time period needed to complete the comparison and selection functions in figure 2.7.3. It was assumed that no line will accept any request during a block transfer operation on that line. Hence there is enough time to update the lookup table for cache misses. A word in the explicit lookup tables contains both a tag and the physical address of a block in the referenced module. In this case, whether an accepted request results in a hit or miss is determined before the cache cycle starts.

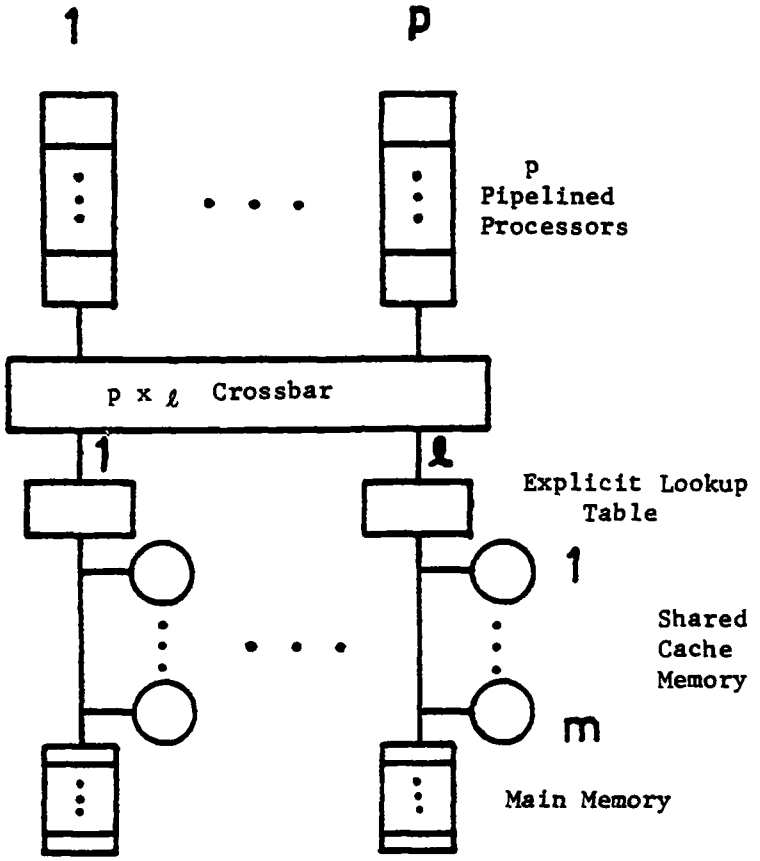


Figure 2.7.4(a) Shared cache with one explicit lookup table per line.

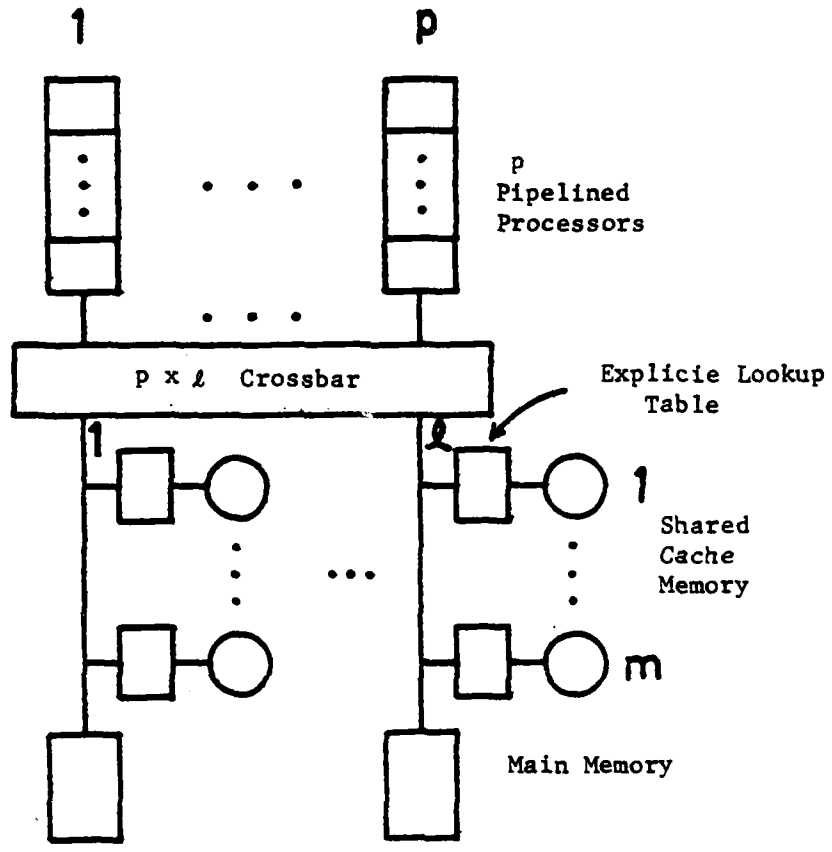


Figure 2.7.4(b) Shared cache with one explicit lookup table per cache module.

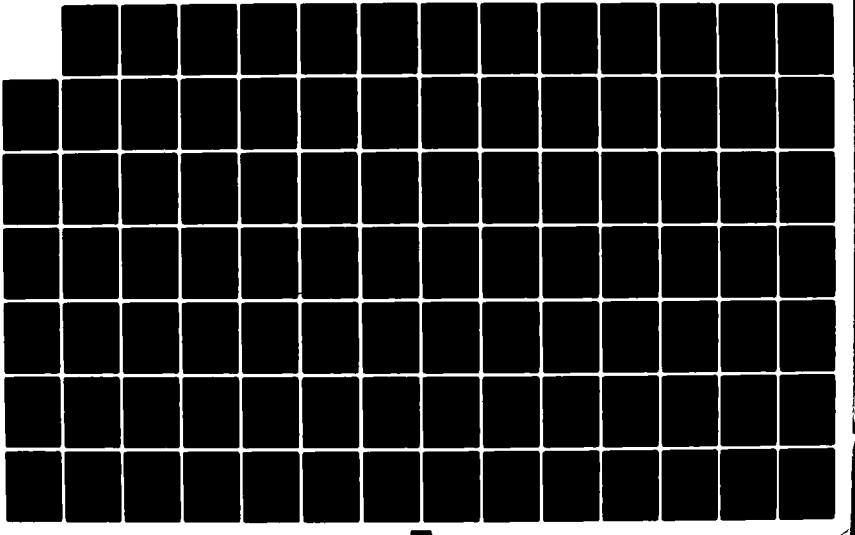
2.8 Concluding Remarks

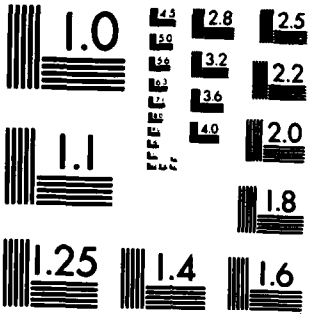
In this chapter, we have discussed cache memory management policies and shared-cache memory organizations. We have shown that a centralized lookup table is not suitable for multiple-stream shared-cache computer systems due to potentially high lookup table access conflicts. We have also shown that a distributed lookup table for fully associative and sector mapping mechanisms may cause a multicopy problem in the local lookup tables and a lookup table maintenance problem. Thrashing and deadlock may occur in direct mapping due to the high possibility of cache block contention among streams. However, set associative mapping allows a distributed lookup table and does not have the above undesirable features. In addition, the set associative mapping mechanism is cost-effective and performs almost as well as fully associative mapping if a sufficiently large set size is used. We suggest that the set associative mapping with an LRU replacement algorithm for each set be used in shared-cache systems.

We have investigated two methods of handling write misses, namely write allocation and no-write allocation. Several schemes of updating the main memory have been discussed. Due to the effect of architectural differences between shared cache and private cache on handling of write-miss and updating the main memory, we suggest that the write-through with buffering updating scheme and the no-write allocation strategy be used in shared cache systems and the write-back with buffering updating scheme and the write allocation strategy be used in private cache systems.

AD-A124 387 SHARED CACHE ORGANIZATION FOR MULTIPLE-STREAM COMPUTER SYSTEMS(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE LAB C YEH JAN 81 R-904 N00039-80-C-0556 2/3

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Furthermore, the application of the L-M memory organization to our shared-cache memory has been discussed. In order to keep the complexity of cache memory mapping and table lookup within reasonable bounds, a memory interleaved by sets has been introduced. Two possible implementations of addressing interleaved by sets have been presented. The choice between these two implementations involves tradeoffs between the block transfer time and the amount of tag storage required.

Finally, we presented a general description of shared-cache memory request scheduling and overall system configurations. The considerations for realistic implementation of some hardware functions, e.g. LRU stacks, are also discussed.

The functional parameters of a shared-cache memory design have been specified for our purposes in this chapter. In chapter 3, the performance of shared-cache memory systems is analyzed for two distinct cache models. A discrete Markov approach and a probabilistic approach are developed for both models. In addition, a probabilistic model for private-cache systems is also developed.

CHAPTER 3

PERFORMANCE ANALYSIS

3.1 Introduction

In this chapter, the analytic models are developed based on certain assumptions about program referencing behavior. The assumptions which are common to all the models of this research are summarized in this section. Any further assumptions made for each particular model are discussed individually as the analysis of each model is performed. The parameters explicitly related to system performance and those used for modeling are also examined and summarized in this section. Analytical models for a shared-cache multiple-stream system with an implicit lookup table are discussed in section 3.2. In section 3.3, the performance analysis of a shared-cache multiple-stream system with explicit lookup tables is carried out. Finally, the performance of a multiprocessor computer system with private cache memories is evaluated in section 3.4.

In a parallel-pipelined processor of order (s,p), we assume that p simultaneous memory requests are issued to the shared cache memory system every segment time unit. For analytical simplicity, it is further assumed that the addresses of the requests are independent and uniformly distributed among the N identical cache memory modules. Therefore, for

the shared-cache with L-M memory organization, the probability that a request addresses some particular module is 1/N. Similarly, since lines are identical and independent, the probability that a request addresses a particular line is 1/ . Since the sequence of requests made by a pipelined processor is formed by interleaving requests from sp instruction streams, each stream in the processor issues one request every s STUs. This interleaving tends to make the randomness assumption of the analytical model more realistic. Also, since the number of segments, s, is greater than or equal to the cache cycle, c, a memory module which is executing a previous request from an instruction stream would have completed its execution when the next request from the same instruction stream arrives. Hence there is no execution overlap between instructions of the same stream and data dependency within a stream will not affect the stream performance.

The assumption of independence and randomness of the reference patterns requires that unsuccessful requests be discarded. The independent request assumption is tested in the simulation models in which real program traces are used. The method of handling unsuccessful requests in the simulation models is to cause the process with an unsuccessful cache memory request to make a non-computing (or null) pass through the processor segments for one instruction cycle and to resubmit the same request the next cycle. In such a case, the process is blocked until its request is satisfied. The simulation results are then compared with the analytical predictions to determine the inaccuracy caused by the random independent request assumption.

Another important assumption made for all analytical models is that the cache hit ratio is independent of cache access conflict. Although cache memory access conflicts indeed affect the reference patterns, the hit ratios should not be disturbed significantly by the access conflicts if a sufficiently large cache size is used. As with the working set concept for a paging system, a block should reside in the cache for a while before it is removed. This assumption separates the cache hit ratio and the cache access conflict into two independent phenomena and simplifies the analytic modeling significantly. The inaccuracy caused by this assumption will also be checked by simulation and discussed in chapter 4.

In summary, the assumptions common to all our analytic models are as follows:

(1) The request sequence consists of independent references.

(2) Processors operate synchronously.

(3) The processor request rate, $\forall$, equals one (implies $s \geq c$).

(4) Cache hit ratio is independent of cache access conflict.

The functional parameters of a shared cache memory design were described in chapter 2. Given this set of functional parameters, the system performance is then dependent on many component parameters which specify the physical sizes of the components. The following list contains the component parameters related to the system performance of a shared-cache multiple-stream computer system.

(1) Block size.

(2) Set size.

(3) Total shared-cache memory size.

(4) Number of pipelined processors, p.

(5) Degree of multiprogramming per pipelined processor, s.

(6) Number of lines within shared cache memory, ℓ .

(7) Number of cache memory modules per line, m.

(8) Cache memory cycle time, c.

(9) Block transfer time, T.

It is very difficult to develop a satisfactory analytic model which contains all of the above nine parameters. However, those nine parameters are not independent of each other and can be classified in the following three categories: (A) those, including items 1, 2, 3, 4 and 5, related to the shared cache hit ratio; (B) those, including items 1, 7 and 8, related to the block transfer time, 9; and (C) those, including items 4, 6, 7, 8 and 9, related to the cache access conflicts. By using the above relationships between parameters, the parameters required to perform the modeling can be reduced to the following six parameters:

(1) Cache hit ratio (h).

(2) Number of pipelined processors (p).

(3) Number of lines within the shared cache memory (ℓ).

(4) Number of cache memory modules per line (m).

(5) Cache cycle time (c).

(6) Block transfer time (T).

Note that the total number of cache modules, N, is equal to ℓ times m. Due to the complexity of the architecture, the relationships between parameters in category (A) are impossible to obtain by a purely analytic approach. Simulation is required to determine those relationships from which the performance differences between alternative configurations can be accurately determined. Therefore, for a given set of values for the component parameters of a shared cache memory, such as block size, set size and so on, hit ratio is a function of the workload environment, such as p, s and program behavior, and can be determined by simulation runs with real program traces used as the input data. For analytical purposes, hit ratio is then left unevaluated and is treated as an independent model parameter. The analytical models are oriented toward developing the probability of acceptance, P_A, for a typical shared cache memory request. The performance measurement, CPU utilization, can easily be derived from P_A and h.

The performance analysis of shared cache memory systems is carried out using either discrete Markov models or probability based theorems. The Markov models belong to a class of time-homogeneous finite-state Markov Chains [58]. The probabilistic models use a conditional probability concept by which bounds on performance can easily be derived. A probabilistic model for multiprocessor systems with private cache memories is also presented in this chapter.

3.2 Shared Cache Memory with an Implicit Lookup Table

A shared cache memory with an implicit lookup table stores the block tags and data together within the cache modules which can be implemented by RAM chips. In this case, whether an accepted request results in a hit or miss is checked after the cache cycle has been completed as illustrated in figure 2.7.3. For simplicity, it is assumed that no cache module can accept any new request before the result of the previously accepted request has been determined. Although the comparators and the selection circuit (see figure 2.7.3) need not be physically built inside the RAM chips, a cache module will be busy in the interval (t,t+c) if it accepts a request at time t. Therefore the cache cycle, c, is the minimum time period required between two successive requests accepted by one cache module. The block transfer time T, which is a function of block size, main memory cycle and cache cycle, is the time period required to fetch a block from main memory to the cache. Since c and T implicitly characterize the speeds of cache and main memory, (c,T) is then defined as the cycle characteristics. Since all lines in an L-M shared cache memory system are identical and independent, a single line model, instead of a total system model, will be sufficient to analyze system performance [41]. This line decomposition technique simplifies the analysis of the system significantly.

A formal definition of the probability of acceptance is given below to clarify further discussion.

Definition 3.1.1 The steady state probability of acceptance, $P_A(c,T,p)$, is the steady state probability that a request issued by a parallel-pipelined processor of order (s,p) will be accepted by an (ℓ,m) memory configuration with cycle characteristics, (c,T). □

It will be seen that s does not affect performance in this model, as long as s is sufficiently large. Furthermore, the model is developed for general ℓ and m.

3.2.1 Discrete Markov Model

Recall that a write-through updating scheme was assumed for the shared cache memory system, no copy back is necessary for a replaced block when a miss occurs. Hence, a cache module can start the block transfer operation immediately after a miss has been detected. The block transfer time, T, in this case is the time period from detecting a miss until the block transfer operation is completed. Therefore, a module and its associated line will be busy for T time units immediately after a miss occurs in this module. The remaining modules on this line cannot accept any new request during this block transfer period because the line is busy.

Since there is no queue assumed in the shared cache memory to buffer the requests which result in cache miss, those requests have to be rejected if the block transfer operation cannot be initiated immediately to serve the cache miss. Furthermore, one might be concerned that a

cache miss on some line should initiate a block transfer operation if and only if all the modules on that line are idle, otherwise the request which results in a miss should be rejected. However, this idea may cause a miss loop which essentially is a deadlock situation. For example, consider two requests k_1 and k_2 which reference modules m_1 and m_2, respectively. Assume that both modules m_1 and m_2 are on the same line and both requests k_1 and k_2 result in cache misses. Suppose k_1 arrives at time t and starts a cycle on m_1. Less than c time units later k_2 arrives and starts a cycle on m_2. When request k_1 is determined to be a miss, module m_2 is busy and request k_1 will be rejected. Assume further that request k_1 will be resubmitted before module m_2 completes its cycle. Therefore, when request k_2 turns out to be a miss, module m_1 is busy again and request m_2 will also be rejected. The same process may continue forever and these two requests lock each other out. In order to avoid the miss loop, a block transfer operation is initiated immediately after a miss is detected.

Note also that there may be some busy modules on a line when a miss occurs on that line. All requests in process within busy modules on a line will be aborted when an earlier request causes a cache miss on that line. These requests are revised to be considered as rejected requests for simplicity of analysis.

Definition 3.2.1.1 The rejected requests corresponding to aborted cache cycles on a line caused by a cache miss on that line are called aborted requests. □

First we develop the line state space of the shared cache memory system for the case p=1. In this section, we assume c> 1 since c=1 is degenerate and trivial.

Definition 3.2.1.2 The module state at time t is

= ϕ(null), if the module is idle at t,

={r}, if the module is busy at t and has been busy for r STUs, where r is an integer such that $1 \leq r \leq c+T-1$. □

If $1 \leq r \leq c-1$, then the module is busy at t for the cache cycle because it accepted a request r STUs ago. However, if $c \leq r \leq c+T-1$, then the module is busy at t for the block transfer operation because it accepted a request r STUs ago which resulted in a cache miss. Observe that the state of a module which accepted a hit request, i.e., a successful request, c STUs ago is ϕ. If a module accepts a request which results in a miss, this module will then be busy for c+T STUs. Since there are m modules on a line, the states of all m modules on the line represent the state of the line.

Definition 3.2.1.3 A line state, $\lambda(t)$, at time t is the set union of all module states at time t for all modules on the line in question. For convenience, the line state is enclosed in "( )". □

Notice that only nonnull states of modules on the line appear explicitly to specify the state of the line. The line state only

identifies whether some module on the line is in each state, and not which particular modules are in which state. Specific module information is not needed due to the uniform and independt request assumption. Furthermore, there can be at most one module on a line in any one particular nonnull state. Thus there are no repeated nonnull module states and a simple set union of module states gives the line state. If there is more than one busy module on the line at time t, the module states are separated by commas. For instance, consider the line state of a line at t which has two busy modules on it, one of which accepted a request one STU ago and the other, two STU ago. The line state at t for this line will be denoted by (1,2). For convenience, the module states of a line state will be listed in ascending order of busy time. Hence (3,1) will be written as (1,3). Moreover, if all modules on a line are idle at t then the line state is denoted by the empty set, ϕ .

Given the state of the line at t, it is necessary to determine the line state at time t+1. To make this determination, it is necessary to understand the change of module states with time. Given the module state of a module at time t, the module state at time t+1 can be evaluated if it is known whether a request is made to and accepted by that module and whether the previously accepted requests on the same line result in hits or misses. Hence for a given module state, the next module state can be obtained as follows.

Definition 3.2.1.4 If the state of a module at t is ϕ , then next module state (at t+1) is

$=\{1\}$, if a request which addressed the module at t was accepted, or

$=\phi$ otherwise; i.e., either no request addressed the module at t, or a request which addressed the module at t was rejected due to a busy line collision. □

Therefore, a module remains in the null state unless it accepts a request at time t whereupon it will become busy and remain busy either during the interval (t,t+c) or during the interval (t,t+c+T) depending on whether this accepted request results in a hit or miss, respectively. For a busy module, the next state can now be evaluated as follows:

Definition 3.2.1.5 Given that the state of a module at time t is $\{r\}$, where r is an integer such that $1 \leq r \leq c+T-1$, the next module state is

$=\{r+1\}$, if $r < c-1$ and no other busy module on the same line detects a miss, or if $c-1 < r < T+c-1$, or if $r=c-1$ and the module in question detects a cache miss, or

$=\phi$, if $r=c+T-1$, or if $r=c-1$ and a hit occurs in the module in question, or if $r < c-1$ and a miss is detected in any other module on the same line. □

Observe that a busy module cycle will be aborted if any other module on the same line detects a miss before this module completes its cache cycle. The next state of an aborted module is then ϕ regardless of its current state. For a non-aborted module, once it accepts a request, it goes through the state sequence $\{1\}, \{2\}, \ldots, \{c-1\}, \phi$, if the accepted request results in a hit; otherwise a miss is detected when the module is

in state c-1 and it goes through the state sequence {1}, {2},...,{c-1}, {c},...,{c+T-1}, ϕ . Hence the maximum utilization of a module is one accepted request per c STUs. Only a module in the module state ϕ can accept a request. It need not be known whether any request addressed the module in order to evaluate the next state of a busy module. Any request made to a busy module is rejected.

Determining the next line state is as straightforward as determining the next module state. However some definitions are needed here to clarify the presentation.

Definition 3.2.1.6 If $\exists$ r $\in \lambda(t)$ such that $c \leq r \leq c+T-1$, then $\lambda(t)$ is a busy line state, otherwise it is an idle line state. □

Definition 3.2.1.7 An idle line state, $\lambda(t)$, is a potential acceptance state if $1 \in \lambda(t)$, otherwise it is a nonacceptance state. Furthermore, an idle line state, $\lambda(t)$, is called a checking state if $c-1 \in \lambda(t)$. □

The request corresponding to the module with state 1 in a potential acceptance state is a potentially accepted request. Whether a potentially accepted request is an accepted request is dependent on whether all the other busy modules on the line are processing hits. If any other busy module in λ results in a miss, this potentially accepted request will be aborted and becomes a rejected request. Therefore, a potentially accepted request is an accepted request if and only if all the other busy modules in λ are processing hits. Obviously, the

potentially accepted request in $\lambda = (1)$ is an accepted request. A checking state can be either a potential acceptance state or a nonacceptance state. At the checking state, a result of hit or miss for the module with state c-1 is detected.

Note that there are at least two possible state transitions from a particular idle line state, $\lambda(t)$: one is to a potential acceptance state and the other is to a nonacceptance state. Three possible state transitions exist only from a checking state: If a hit results, the state may go to either of the two next idle line states mentioned above; otherwise it goes to the next busy line state $\lambda = (c)$. Only two posssible state transitions exist from nonchecking idle line states. For a busy line state, the line is busy for the block transfer operation and will only make a transition to its successor busy line state (or the null state if $\lambda = (T+c-1)$) whether or not a request addresses the line.

In order to develop the Markov model required to analyze the shared cache memory conflict problem on a line, the line state space is investigated. However, we need to know the probability of transition from one state to another in order to compute the probability of being in each state. The cardinality of a state is useful in obtaining the transition probabilities.

Definition 3.2.1.8 The number of elements or the cardinality of a line state $\lambda(t)$, $|\lambda(t)|$, is the number of nonnull module states in the line state. □

Note that the cardinality of a busy line state is one. Since all the aborted modules on one line have the module state ϕ and will not be listed, only the module which causes the cache miss will be listed in the line state. Although the m modules on one particular line may be all involved in a block transfer operation at the same time if the implememtation of figure 2.5.2(b) is used, a busy line state with one element is sufficient to describe the line state because none of the m modules on the line can accept any new request during the block transfer operation.

Recall that when a request references an idle module on an idle line, the request is potentially accepted, causing the line state to make a transition to a potential acceptance line state. Similarly, a line state makes a transition to a nonacceptance line state if no request referenced the line or a request which referenced the line was rejected. The transition probabilities can then be obtained with the aid of the following definition and theorems.

Definition 3.2.1.9 The probability of transition, $p_{i,j}$, is the conditional probability of going from a given line state λ_i , at time t, to its successor line state λ_j , at time t+1. Rewriting this statement in probability notation, $p_{i,j}=p(\lambda_j / \lambda_i)$. □

Theorem 3.2.1.1 The probability of a request being rejected due to multiple access line collision with one or more of the p-1 other simultaneous requests is

$$P_1 = 1 - [\ 1 - (\ 1 - \frac{1}{\ell}\ )^p\ ]\ \frac{\ell}{p}$$

Proof: Since there are ℓ lines in the shared cache memory system, a request will reference a particular line with probability $1/\ell$. Thus $(1-1/\ell)^p$ is the probability that no request references a particular line and $1-(1-1/\ell)^p$ is the probability that there is at least one request to a particular line. The expected number of distinct lines referenced by p requests, i.e. the line bandwidth, is then $[1-(1-1/\ell)^p]\ell$. The probability P_1 is then 1-line bandwidth/p. □

Note that $1-P_1$ is a closed form representation of Ravi's results[37]. Chang [59] showed the equivalence of $1-P_1$ and Ravi's result. Strecker [19] derived the same result by using a different approach. Briggs [41] also proved this result.

Lemma 3.2.1.1.1 The cache memory request rate seen by a particular line in the shared cache is

$$q = \frac{p(1-P_1)}{\ell} = 1 - (1-\frac{1}{\ell}\ )^p$$

Proof: From theorem 3.2.1.1, it is clear that $p(1-P_1)$ requests pass through the crossbar every STU. These $p(1-P_1)$ requests are the requests seen by the ℓ lines. Since they all reference different lines and all

lines are identical, the probability that a particular line will be referenced by one of these requests, i.e. the request rate seen by this particular line, is $p(1-P_1)/\ell = 1-(1-1/\ell)^p$. □

Theorem 3.2.1.2 The probability of transition from an idle line state (t) to its successor potential acceptance state is

$$P_a(\lambda) = \frac{m-|\lambda|}{m} qh, \qquad \text{if } c\text{-}1 \in \lambda$$
$$= \frac{m-|\lambda|}{m} q, \qquad \text{otherwise}$$

Where $|\lambda|$ is the cardinality of the line state $\lambda(t)$.

Proof: Suppose that the idle line state, $\lambda(t)$, is not a checking state. Then request which addresses a line in state $\lambda(t)$ will be potentially accepted if the request addresses an idle module on the line. The number of idle modules on the line represented by $\lambda(t)$ is $m-|\lambda|$, where $|\lambda|$ is the number of busy modules on the line. Given a request to the line, the conditional probability of requesting any one of the idle modules on the line is $(m-|\lambda|)/m$. Since the probability of requesting the line is q, the probability of a request referencing some idle module on a particular idle line is $q[(m-|\lambda|)/m]$. Therefore, $P_a=(m-|\lambda|)q/m$ if the referenced idle line state, $\lambda(t)$, is not a checking state. □

Otherwise, $\lambda(t)$ is a checking state and $P_a=[(m-|\lambda|)/m]hq$ because a request is potentially accepted at a checking state if and only if the

module with state c-1 results in a hit and the conditions above also apply.

Corollary 3.2.1.2.1 The probability of transition from a checking state, $\lambda(t)$, to its successor busy line state is

$$P_b(\lambda) = 1 - h$$

Proof: By definition, an idle line state makes a transition to its successor busy line state if and only if the idle line state, $\lambda(t)$, is a checking state and a miss results. □

Theorem 3.2.1.3 The probability of transition from an idle line state, $\lambda(t)$, to its successor nonacceptance state is

$$P_n(\lambda) = h - P_a(\lambda), \quad \text{if } c\text{-}1 \in \lambda$$

$$= 1 - P_a(\lambda), \quad \text{otherwise}$$

Proof: Note that $P_a(\lambda)+P_n(\lambda)=1$ for all the idle line states except the checking states. For a nonchecking idle line state, nonacceptance implies either rejection of a request to the line or no request arrival to that line. For the checking states, $P_a(\lambda)+P_b(\lambda)+P_n(\lambda)=1$. A checking state makes a transition to its successor busy line state if a miss results, a transition to its successor nonacceptance state or its successor potential acceptance state if a hit results. Since $P_b(\lambda)=1-h$,

$P_n(\lambda) = h - P_a(\lambda)$ follows. □

Theorem 3.2.1.4 The probability of transition from a busy line state, $\lambda(t)$, to its successor busy line state (or to ϕ if $\lambda(t) = (T+c-1)$) is one.

Proof: Since $\lambda(t)$ is a busy line state, there is no successor potential acceptance state hence the result follows.

For given cycle characteristics, (c,T), a line state diagram is readily constructed by following the above definitions and transition probabilities. For example, a line state diagram for cycle characteristics $(c,T)=(3,10)$ and $m \geq 2$ is shown in figure 3.2.1.1, where * indicates the potential acceptance states. □

Theorem 3.2.1.5 The total number of distinct line states for cycle characteristics (c,T) is

$$\text{Total number of distinct line states} = \begin{cases} 2^{c-1} + T, & \text{for } m \geq c-1 \\ \sum_{0 \leq i \leq m} \binom{c-1}{i} + T, & \text{for } m < c-1 \end{cases}$$

Proof: For $m \geq c-1$, every idle line state except the null line state and the checking states can generate two new successor states: one is the next nonacceptance state and the other is the next potential acceptance

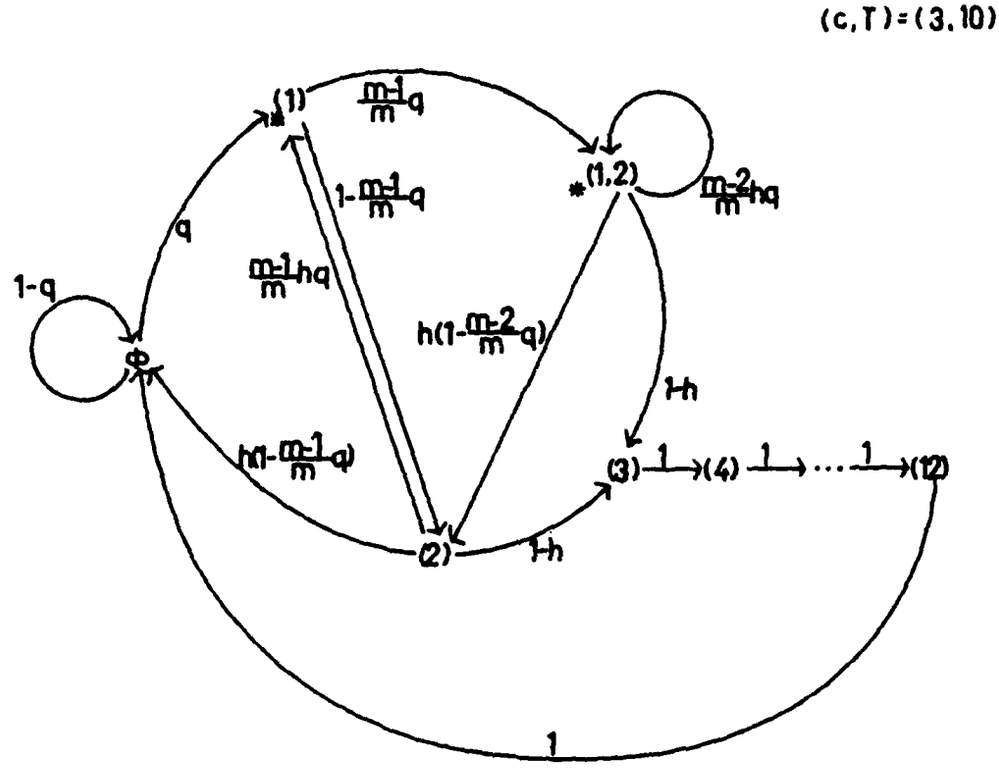


Figure 3.2.1.1 Line state diagram for shared cache with an implicit lookup table and cycle characteristics (c,T) = (3,10).

state. Therefore, the idle line states excluding the null line state actually form a binary tree structure with tree height c-2. The number of states contained in this binary tree is $2^{c-1}-1$. (T+1) accounts for the T busy line states plus one null line state. Hence the total number of distinct states is $2^{c-1}+T$ for $m < c-1$. The number of busy line states is still T for $m \geq c-1$. However, for $m < c-1$, there are at most m busy modules in each idle line state. Therefore, the total number of distinct idle line states is $\sum_{0 \leq i \leq m} \binom{c-1}{i}$. □

The probability that a particular line accepts a request, denoted as $P_{A\ell}(c,T,p)$, can be found by solving the Markov model developed in this section. The following theorems help us to evaluate the system probability of acceptance, $P_A(c,T,p)$, for the shared-cache L-M memory organization with cycle characteristics (c,T).

Theorem 3.2.1.6 For the L-M shared cache memory the steady state probability of acceptance, $P_{A\ell}(c,T,p)$, that a particular line at a particular STU accepts a request is

$$P_{A\ell}(c,T,p) = \sum_{\lambda \in \{\lambda \mid 1 \in \lambda\}} h^{|\lambda|-1} p_\lambda,$$

where p_λ is the probability of being in line state λ.

Proof: A potentially accepted request is an accepted request if all the other busy modules on the same line are processing hits, otherwise this potentially accepted request will be aborted. Therefore, the probability that a particular potential acceptance state, $\lambda(t)$, accepts a request is the probability of being in that state times the probability that all the other busy modules on the same line result in hits. The latter probability is simply $h^{|\lambda|-1}$, where $|\lambda|$ is the cardinality of the line state $\lambda(t)$. Then, $P_{A\ell}(c,T,p)$ is this probability of accepting a request summed over all the potential acceptance states of the line being modeled. □

Theorem 3.2.1.7 The steady state probability of acceptance of a particular request in the L-M shared cache memory organization is

$$P_A(c,T,p) = \ell P_{A_\ell}(c,T,p)/p$$

Proof: All the ℓ lines of the L-M shared cache memory organization are identical and independent. Hence $\ell P_{A\ell}(c,T,p)$ is the expected total number of the accepted requests per STU, i.e. $pP_A(c,T,p)$. □

Note that the line state diagram generated by the Markov model developed in this section is an ergodic Markov Chain because every line state can be reached from any other line state. Therefore, the existence of a unique equilibrium solution for this model is guaranteed [58].

Corollary 3.2.1.7.1 The probability of success of a particular request in the L-M shared cache memory organization is

$$P_S(c,T,p) = h\, P_A(c,T,p) \qquad \square$$

Obviously a request is successful if the request is accepted and results in a hit.

The example of figure 3.2.1.1, (c,T)=(3,10), can then be solved as follows.

Let P_λ denote the probability of being in the line state λ. There are only two potential acceptance states shown in figure 3.2.1.1, namely, state (1) and state (1,2).

Hence $P_{A\ell}(c,T,p) = P_{A\ell}(3,10,p) = p_1 + hp_{1,2}$

From figure 3.2.1.1, the following equations can easily be obtained.

$$p_\phi = (1-q)p_\phi + h\left(\frac{m-mq+q}{m}\right) p_2 + p_{12} \tag{1}$$

$$p_1 = qp_\phi + \frac{mhq-hq}{m}\, p_2 \tag{2}$$

$$p_2 = \frac{m-mq+q}{m}\, p_1 + h\left(\frac{m-mq+2q}{m}\right) p_{1,2} \tag{3}$$

$$p_3 = (1-h)\,[p_2 + p_{1,2}] \tag{4}$$

$$p_{1,2} = \frac{mq-q}{m}\, p_1 + \frac{mhq-2hq}{m}\, p_{1,2} \tag{5}$$

From equation (5)

$$\left(\frac{m-mhq+2hq}{m}\right) p_{1,2} = \frac{mq-q}{m}\, p_1$$

$$P_{1,2} = \frac{mq-q}{m-mhq+2hq} P_1$$

From equation (2)

$$qP_\phi = P_1 - \left(\frac{mhq-hq}{m}\right)\left(\frac{m-mq+q+hq}{m-mhq+2hq}\right)P_1$$

$$P_\phi = \frac{(m^2-2m^2hq+3hqm-2mhq^2+m^2hq^2-mh^2q^2+hq^2+h^2q^2)p_1}{mq(m-mhq+2hq)}$$

Note that $P_\phi + P_1 + P_2 + P_{1,2} + TP_3 = 1$

Solve for P_1

$$P_1 = \frac{qm(m-mhq+2hq)}{(1-2hq+2q+qT-Thq)m^2+(3hq-h^2q^2+hq^2+Thq^2-Th^2q^2)m+hq^2+h^2q^2}$$

$$P_1 = \frac{\ell qN(N-Nhq+2h\ell q)}{(\ell-2h\ell q+2\ell q+\ell qT-Th\ell q)N^2+(3h\ell^2q-\ell^2h^2q^2+h\ell^2q^2+Th\ell^2q^2-Th^2\ell^2q^2)N+\ell^3hq^2+\ell^3h^2q^2}$$

$$P_1 = \frac{\ell qN(N-Nhq+2h\ell q)}{\Delta}$$

From equation (3)

$$P_2 = \left[\frac{m-mq+q}{m} + \frac{(mh-mhq+2hq)(mq-q)}{m(m-mhq+2hq)}\right] P_1$$

$$P_2 = \frac{m-mq+q+hq}{m-mhq+2hq} P_1$$

From equation (4)

$$P_3 = (1-h)\left[\frac{m-mq+q+hq}{m-mhq+2hq} + \frac{mq-q}{m-mhq+2hq}\right] P_1$$

$$P_3 = \frac{(1-h)(m+hq)}{m-mhq+2hq} P_1$$

Since $P_{A\ell} = p_1 + hp_{1,2} = \dfrac{\ell qN(N+h\ell q)}{\Delta}$,

and $\ell P_{A\ell} = pP_A$,

then $P_A = \dfrac{\ell(1-P_1)N(N+h\ell q)}{\Delta}$,

where $\Delta = (\ell+12\ell q-12h\ell q)N^2+(3h\ell^2 q-11h^2\ell^2 q^2+11h\ell^2 q^2)N+\ell^3 q^2 h(1+h)$,

and $(1-P_1) = [1-(1-1/\ell)^p]\ell/p$.

Only one parameter, c, instead of both c and T, needs to be given in order to construct the Markov state diagram and obtain the closed form solutions. For example, if the state (12) in figure 3.2.1.1 is replaced by (c+T-1)=(T+2), then a line state diagram for (c,T)=(3,T) is obtained. By going through the same computation process, a closed form solution which contains T as a parameter can be obtained.

As can be seen from theorem 3.2.1.5, the number of states in a line state diagram increases with both c and T. The total number of distinct line states increases linearly with T but exponentially with c. Therefore, for a large value of c, it is computationally tedious to obtain the exact solution. In practice, cache speed should be very fast in order to match the processor speed. Hence reasonably small values of c should be sufficient to model all important cases.

3.2.2. Probabilistic Model

No closed form solution for $P_A(c,T,p)$ exists for general (c,T). We

must know the value of c in order to solve the Markov state diagram for the probability of acceptance, $P_A(c,T,p)$. The technique used requires the computation of the steady state probability of being in certain line states. In this section, an alternative approach, which can give more insight into the effects caused by individual blocking conditions, is discussed.

In chapter 2, it was shown that a request may be blocked for one of four different reasons. For convenience, those four reasons are repeated here. A request made to the shared cache memory system may be blocked due to

(1) Multiple access line collision (only if $p > 1$),
(2) Busy line collision (only if $h < 1$ and $T > 0$),
(3) Busy module collision (only if $c > 1$), or
(4) Cache miss (only if $h < 1$).

Recall that the aborted requests, introduced in the last section, are considered as rejected requests. For conceptual simplicity, the rejections of those aborted requests are classified as rejections due to busy line collision in the following discussion. Therefore, the busy line collision actually includes the rejections of a request referencing a busy line and being aborted by a miss of an earlier request on the same line. Since the possibility of a request being aborted is considered in the busy line collision, the rejection due to busy module collision is only applied to nonaborted requests. In other words, not every busy

module is affected by the busy module collision. Those busy modules on an idle line will not be affected by the busy module collision if any one of them results in a miss since an incoming request will be aborted in this case. Hence only those busy modules on an idle line all of whose busy modules result in hits affect the busy line collision. For convenience, those busy modules which affect busy module collisions are called blocking modules. By this modification of the previous definitions of busy line and busy module collisions, the above four blocking factors can still be applied in the following discussion without defining new terminology.

Let P_1, P_2, P_3 and P_4 be the probabilities of blocking of a request due to the above four events respectively. Then the probability of a request being blocked by the shared cache memory system can be obtained by considering mutually exclusive and independent blocking events.

Theorem 3.2.2.1 The probability of blocking a request issued to the L-M shared cache memory system whose cycle characteristics are (c,T) is

$$P_B=P_1+(1-P_1)P_2+(1-P_1)(1-P_2)P_3+(1-P_1)(1-P_2)(1-P_3)P_4 \qquad \square$$

Notice that $1-P_1$ is the probability that blocking does not occur due to event i.

Corollary 3.2.2.1.1 The probability of success of a request made to the L-M shared cache memory whose cycle characteristics are (c,T) is

$$P_S(c,T,p)=1-P_B(c,T,p) \qquad \square$$

Obviously an accepted request results in a cache miss with probability 1-h. Hence $P_4=1-h$ and $P_S= h(1-P_1)(1-P_2)(1-P_3)$. Note that P_1 is given in Theorem 3.2.1.1. The probability of acceptance, $P_A(c,T,p)$, is then $(1-P_1)(1-P_2)(1-P_3)$ by corollary 3.2.1.7.1. For brevity $P_A(c,T,p)$, $P_B(c,T,p)$ and $P_S(c,T,p)$ will sometimes be written as P_A, P_B and P_S respectively.

A request will be rejected due to busy line collision if it has no multiple access line collision, but references a busy line.

Lemma 3.2.2.1 The probability of a request referencing a busy line (or being aborted by a previous miss) is

$$P_2 = \frac{(T+c-1)(1-h)pP_A}{\ell}$$

Proof: A potentially accepted request will be aborted by any other busy module which results in a miss on the same line while the potentially accepted request is still in process. This rejection is considered as a busy line rejection. If a busy module with the checking module state, c-1, results in a miss, this module will not only cause the line to be busy for the following T time units but will also cause rejection of all the requests potentially accepted by that line during the last c-1 time units. Therefore, a cache miss occuring on a line actually has the effect of blocking all requests to that line for T+c-1 time units. In

other words, assuming that the line busy checker has the ability to look forward such that when the line first accepts a request which will result in a miss, the line becomes busy immediately after accepting the request and remains busy for T+c time units. Therefore, the expected number of busy lines, E(BL), is the expected number of accepted requests which will result in misses over a period of T+c-1 time units. Under the independent request assumption, we have $E(BL) = (T+c-1)(1-h)pP_A$. The probability, $P_2 = E(BL)/\ell$. □

Corollary 3.2.2.1.2 The expected number of idle lines is

ℓ_{idle} = ℓ -(expected number of busy lines)

= ℓ $-(T+c-1)(1-h)pP_A$

= ℓ $(1-P_2)$. □

The computation of the probability of referencing a blocking module on an idle line, P_3, is not always straightforward. However, P_3 can be generalized by the next Lemma.

Lemma 3.2.2.2 The probability of referencing a blocking module on an idle line is

$$P_3 = \frac{E(BM/IL)}{E(M/IL)}$$

Where E(BM/IL) is the expected number of blocking modules on idle lines

and E(M/IL) is the expected number of modules on idle lines. Notice that $E(M/IL)=\ell_{idle}m=\ell m(1-P_2)=N(1-P_2)$. □

The derivation of E(BM/IL) for given (c,T) can be made from the line state diagrams. Since all ℓ lines are identical and independent, the line state diagrams for all ℓ lines are identical and independent. Hence we can model the entire system by modeling one line. At steady state, if p requests are issued, pP_A requests are accepted by the system. These accepted requests cause the addressed lines to make transitions to potential acceptance states. Since the request references are uniformly distributed over the ℓ lines, the accepted requests will be uniformly distributed over all ℓ lines. The following definitions are made to aid in the derivation of E(BM/IL) for the system.

Definition 3.2.2.1 For the system, $E(\lambda)$ is the expected number of lines at any time instant which are in the line state λ. □

Definition 3.2.2.2 S(c,T) is the set of nonnull idle line states in the line state diagram for cycle characteristics (c,T), i.e. $S(c,T) = \{ \lambda \mid \lambda \neq \phi$ and if $r \in \lambda$ then $r < c\}$ □

Lemma 3.2.2.3 At the steady state, the expected number of accepted requests per STU of a whole system with cycle characteristics (c,T) is

$$pP_A = \sum_{\lambda \in \{\lambda | 1 \in \lambda\}} h^{|\lambda|-1} \; E(\lambda)$$

Proof: Multiplying the equation given in theorem 3.2.1.6 by ℓ gives

$$\ell P_{A\ell(c,T,p)} = \sum_{\lambda \in \{\lambda | 1 \in \lambda\}} h^{|\lambda|-1} \; \ell p_\lambda$$

Then, substituting $E(\lambda)$ for ℓp_λ, using Definition 3.2.2.1, and pP_A for $\ell P_{A\ell}$, using Theorem 3.2.1.7, this lemma follows immediately. □

Lemma 3.2.2.4

$$E(BM/IL) = \sum_{\lambda \in S(c,T)} (h^{|\lambda|} \; E(\lambda) \; |\lambda|)$$

Proof: The expected number of modules on lines with nonnull idle line states is

$$\sum_{\lambda \in S(c,T)} (E(\lambda) \; |\lambda|)$$

However, some of those modules may not affect busy module collisions. Only blocking modules affect busy module collisions. The busy modules on a line are blocking modules if and only if they all result in hits. Otherwise, an incoming request to this line will be aborted by some busy module which results in a miss on this line. Thus $h^{|\lambda|}$ must be used as a multiplicative factor in the above formula. □

The probability of acceptance, $P_A(c,T,p)$, of the cycle characteristics (3,10) can readily be obtained by using the probabilities for the four different blocking factors developed in this section.

From figure 3.2.1.1 and lemma 3.2.2.3

$$E(1) + hE(1,2) = pP_A \qquad (1)$$

Let p_λ denote the steady state probability of being in state λ. Since $\ell p_\lambda = E(\lambda)$, then the relationships between p_i and p_j are also the relationships between E(i) and E(j).

Therefore,

$$E(1,2) = \frac{mq-q}{m-mhq+2hq} E(1)$$

and

$$E(2) = \frac{m-mq+q+hq}{m-mhq+2hq} E(1)$$

Substituting for E(1,2) in equation (1),

$$E(1) = \frac{m-mhq+2hq}{m+hq} pP_A$$

From lemma 3.2.2.4

$$E(BM/IL) = hE(1) + hE(2) + 2h^2E(1,2) \qquad (2)$$

Substituting for E(1), E(2) and E(1,2) in E(BM/IL),

$$E(BM/IL) = \frac{2hm-mhq+hq+h^2q+h^2mq}{m+hq} pP_A$$

From lemma 3.2.2.2

$$P_3 = \frac{E(BM/IL)}{N(1-P_2)}$$

$$P_3 = \frac{2hm-mhq+hq+h^2q+h^2mq}{N(1-P_2)(m+hq)} pP_A$$

From lemma 3.2.2.1

$$P_2 = \frac{(T+c-1)(1-h)pP_A}{\ell}$$

$$= \frac{12(1-h)pP_A}{\ell}$$

Substituting for P_2 and P_3 in

$$P_A = (1-P_1)(1-P_2)(1-P_3)$$

and solving for P_A, we have

$$P_A = \frac{qm(m-mhq+2hq)+qmh(mq-q)}{(1-12qh+12q)m^2+(3hq-11h^2q^2+11hq^2)m+hq^2+h^2q^2}$$

$$= \frac{\ell(1-P_1)N(N+h\ell q)}{\Delta}$$

where $\Delta = (\ell+12\ell q-12h\ell q)N^2+(3h\ell^2q-11h^2\ell^2q^2+11h\ell^2q^2)N+\ell^3q^2h(1+h)$.
Note that the result above is identical to the result obtained in last section.

Given a shared cache memory organization and a workload, the hit ratio, h, of such a system can be obtained from simulation. The probability of acceptance, $P_A(c,T,p)$, can be obtained from the analyses developed in these last two sections. The performance measurement, i.e., CPU utilization, is then given as follows.

Theorem 3.2.2.2 The CPU utilization, Cu, for a shared cache memory with implicit lookup table is

$$Cu = \frac{1}{\frac{1}{P_A} + (1-h)T''}$$

where $T'' = \left\lceil \frac{T}{s} \right\rceil$.

Proof: Let T" be the block transfer time relative to pipelined processor cycle time, s, i.e. $T''=\lceil T/s \rceil$. Recall that an unsuccessful request will cause the corresponding processor to make a null pass, i.e. a noncomputing pass, through one cycle. Hence the total number of null passes for each satisfied (successful) request is

$$P_A(1-h)T''+(1-P_A)P_A[1+(1-h)T'']+(1-P_A)^2P_A[2+(1-h)T'']+\ldots.$$

$$=(1-h)T''P_A[1+(1-P_A)+(1-P_A)^2+\ldots]+(1-P_A)P_A[1+2(1-P_A)+3(1-P_A)^2+\ldots]$$

$$=(1-h)T''P_A\left(1-\frac{1}{1-P_A}\right)+(1-P_A)P_A\left\{\frac{d}{d(1-P_A)}[1+(1-P_A)+(1-P_A)^2+\ldots.]\right\}$$

$$=(1-h)T''+(1-P_A)P_A\frac{d}{d(1-P_A)}\left[\frac{1}{1-(1-P_A)}\right]$$

$$= (1-h)T'' + \frac{1-P_A}{P_A}$$

$$= (1-h)T'' + \frac{1}{P_A} - 1$$

where $(\frac{1}{P_A} - 1)$ is the penalty for the access conflicts and $(1-h)T''$ is the penalty for a cache miss. Therefore, the total number of passes a request must take is

$$(1-h)T'' + (\frac{1}{P_A} - 1) + 1 = \frac{1}{P_A} + (1-h)T''$$

Thus $$Cu = \frac{1}{\frac{1}{P_A} + (1-h)T''}$$ □

Note that this formula does not consider the situation in which processors have to make an extra request to obtain the data from cache after a block transfer operation has been completed. However, this formula is still applicable to the system in which an extra request is required because h can be adjusted. In fact, for a high performance system, i.e. high hit ratio, the performance difference caused by this one extra request is negligible.

The probability of acceptance, $P_A(c,T,p)$, for the cases c=1, 2, and 3, is listed in Appendix A. For $c > 3$, the computation becomes enormous and the result is extremely complex. In the following section, simpler upper and lower bounds for $P_A(c,T,p)$ are derived to provide a rough prediction of the performance.

3.2.3 Bounds on $P_A(c,T,p)$

It was seen in last two sections that obtaining $P_A(c,T,p)$ for large values of c is a formidable problem. However, upper and lower bounds can be obtained for $P_A(c,T,p)$. These give a rough estimate for design purposes.

Theorem 3.2.3.1 An upper bound on the expected number of blocking modules on idle lines for a given (ℓ,m) is

$$(c-1)pP_S$$

Proof: Here $(c-1)pP_S$ is the total number of successful requests during last c-1 time units, i.e. the total number of busy modules. However not every busy module resulting from a successful request is a blocking module. For a line state that contains more than one nonnull element, an incoming request will be aborted due to a miss occuring in any one of the busy modules in this line state. This rejection is considered as a busy line collision. The probability of this rejection is included in P_2, not in P_3. Nevertheless $(c-1)pP_S$ includes those busy modules and therefore overestimates E(BM/IL). □

For example, the expected number of blocking modules on idle lines for c=3 is

$$E(BM/IL)=hE(1)+hE(2)+2h^2E(1,2) \qquad (1)$$

By Lemma 3.2.2.3

$$E(1)+hE(1,2)=pP_A$$

Then $hE(1)+h^2E(1,2)=pP_S$

In the steady state, the total number of successful requests during each STU is equal to the total number completed cache memory cycles resulting in hits.

Hence

$$hE(2)+hE(1,2)=pP_S$$

By theorem 3.2.3.1 the upper bound of E(BM/IL) is $(c-1)pP_S=2pP_S$

and adding the two equations above,

$$2pP_S=hE(1)+hE(2)+hE(1,2)+h^2E(1,2)$$
$$=hE(1)+hE(2)+2h^2E(1,2)+(1-h)hE(1,2) \qquad (2)$$

The difference between equation (1) and equation (2) is the last term, i.e., $(1-h)hE(1,2)$, in equation (2). This term shows that one blocking module is counted in the upper bound for state (1,2) when one module is processing a hit and the other a miss. Since the request of module in state 1 cannot be accepted if module in state 2 is processing a miss, the module in state 1 must be processing a miss and the module in state 2 a hit. However, the nonnull modules on a line are blocking modules if and only if they are all hits. Hence $(1-h)hE(1,2)$ is the amount of overestimation made by $(c-1)pP_S$.

Note that an idle line state $\lambda(t)$ has $|\lambda|$ blocking modules if and only if those $|\lambda|$ nonnull elements all result in hits, otherwise no busy module on this line is counted in E(BM/IL).

Corollary 3.2.3.1.1 For $c \leq 2$ or $m=1$,

$$E(BM/IL)=(c-1)pP_S \qquad \square$$

For $c\leq 2$ or $m=1$, there is no line state with more than one nonnull element. Therefore, no request will be aborted by any other request on the same line and no overestimation of E(BM/IL) will be made by $(c-1)pP_S$.

Corollary 3.2.3.1.2 A lower bound on the probability of acceptance for a given (ℓ,m) is

$$P_A \geq \frac{\ell N(1-P_1)}{\ell N + Np(1-P_1)(1-h)(T+c-1) + \ell ph(1-P_1)(c-1)}$$

where $P_1 = 1-[1-(1-\frac{1}{\ell})^p]\frac{\ell}{p}$. $\square$

This Corollary follows directly by substituting $(c-1)pP_S$ for E(BM/IL) in P_3 and then plugging the formulas for P_2 and P_3, i.e. lemmas 3.2.2.1 and 3.2.2.2 respectively, into $(1-P_1)(1-P_2)(1-P_3)$.

Theorem 3.2.3.2 The maximum performance memory configuration for any (c,T) is $(\ell,m)=(N,1)$ and for this configuration

$$P_A = \frac{N(1-P_1)}{N + p(1-P_1)[T(1-h) + (c-1)]}$$

Proof: It is trivial to show, since increasing ℓ cannot decrease performance, that the maximum performance memory configuration is

$(\ell,m)=(N,1)$. Figure 3.2.3.1 shows the line state diagram for $\ell=N$. Notice when $\ell=N$ that m=1 and an idle line cannot accept a request unless the module on that line is idle. From corollary 3.2.3.1.1, $E(BM/IL)=(c-1)phP_A$. Hence $P_3 = (c-1)phP_A/N(1-P_2)$. Then plugging the formulas for P_2, i.e. lemma 3.2.2.1, and for P_3, i.e. lemma 3.2.2.2, into the equation $P_A = (1-P_1)(1-P_2)(1-P_3)$ and substituting N for ℓ gives

$$P_A(c,T,p) = \frac{N(1-P_1)}{N + p(1-P_1)[T(1-h) + (c-1)]} . \qquad \square$$

Although $(\ell,m)=(N,1)$ is the maximum performance configuration, this configuration is often undesirable for large N because of the cost incurred by the increased crossbar size, p x ℓ.

Corollary 3.2.3.2.1 The minimum performance memory configuration for any (c,T) is $(\ell,m)=(1,N)$ and the lower bound $P_A(c,T,p)$ for this configuration is

$$P_A \geq \frac{N}{p[N(T+c-Th) - h(c-1)(N-1)]}$$

Proof: It is also trivial to show that the minimum performance memory configuration is $(\ell,m)= (1,N)$. Using $E(BM/IL)=(c-1)phP_A$ in P_3, this corollary can easily be obtained by plugging the formulas for P_2 and P_3 into the equation $P_A = (1-P_1)(1-P_2)(1-P_3)$ and substituting N for m. $\square$

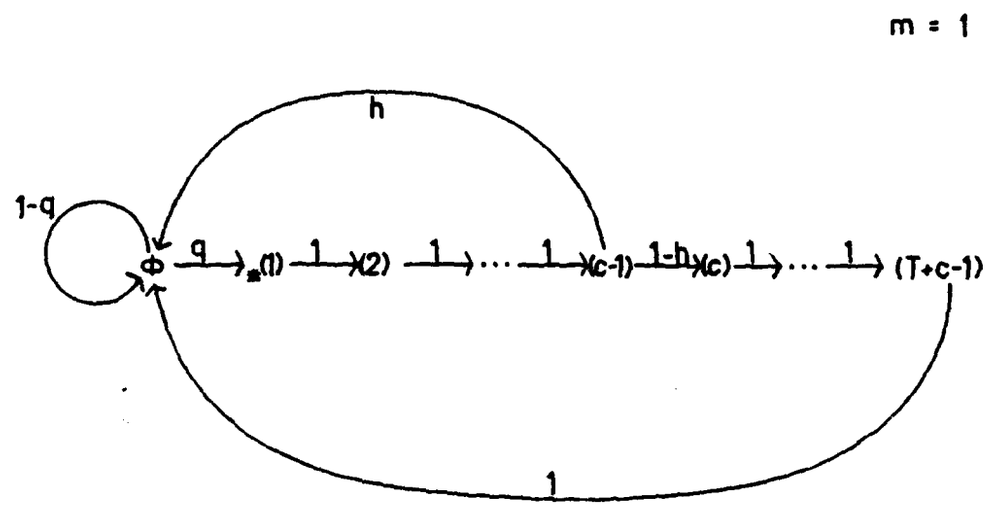


Figure 3.2.3.1 Line state diagram for shared cache with an implicit lookup table and m = 1.

3.3 Shared Cache Memory with Explicit Lookup Tables

A shared cache memory with explicit lookup tables stores the block tags and data separately in distinct memory units. The lookup tables contain not only the block tags but also the physical block addresses and the specific cache modules to which they refer. CAM chips are usually used to implement these lookup tables because of the parallel associative search required. The cache module itself can be implemented by RAM devices. As stated in section 2.7, part of the explicit lookup table can be associated with each line or each module. Both these organizations give the same performance if the cycle time of the lookup table is transparent. The organization of figure 2.7.4(a), i.e., an explicit lookup table associated with each line, is chosen for analytical modeling because of its cleanness of presentation. Note that in this case, whether a request results in a hit or miss is determined before the actual cache cycle is initiated. Hence only hit requests need initiate normal cache cycles.

For this model, a different cache request scheduling strategy is assumed. A request which results in a miss will initiate the block transfer operation if and only if all the cache modules on the line are idle, i.e., the line is at the null state, otherwise this request will be rejected. In other words, the hit requests have higher priority for being served by the shared cache memory than the requests which cause misses. This request scheduling will not cause the miss loop situation as discussed in section 3.2.1 because a request is known to be a hit or miss before any later request can be accepted by the same line.

Therefore, all the accepted requests will finish their cache cycles without being aborted and every busy module is a blocking module. The block transfer time, T, in this case is the time period from when a miss is detected on a line in the null state until the block transfer operation is completed. All the currently busy modules on the line have to complete their cache cycles before a block transfer operation can be initiated on the line. A request resulting in a miss to a line which has at least one busy module on it, is simply rejected. Therefore, a cache module will be busy in the interval (t,t+c) if it accepts a hit request at time t and a line will be busy in the interval (t,t+T) if a miss occurs on a line with no busy module on it at time t.

The analytical approaches used to model the shared cache memory with an implicit lookup table will also be applied to the shared cache memory with explicit lookup tables. Some definitions have to be modified and some results have to be rederived. Those definitions and results developed in previous sections which still can be applied in this case will not be repeated in the following discussions. Let $P_{Ah}(c,T,p)$ denote the probability of acceptance for hit requests, i.e. the fraction of all hit requests which are accepted and $P_{Am}(c,T,p)$ similarly be the probability of acceptance for miss requests, i.e. the probability that a miss request initiates the block transfer operation.

Since a miss request can be accepted only if the referenced line is in the empty state, a miss request is more likely to be rejected by cache memory than a hit request. This unequal probability of rejection for miss and hit requests causes a biased hit ratio seen by the cache memory

system. The effect of this biased request scheduling on hit ratio is discussed in section 3.3.3.

3.3.1 Discrete Markov Model

Again, in this section we assume c >1 since c=1 is degenerate and trivial. Also, it is assumed that the cycle time of the lookup table is transparent. Thus, the lookup tables do not pose a limiting constraint and are not explicitly modeled in this section.

Definition 3.3.1.1 A module state at time t is

$= \phi$(null), if the module is idle at t,

$=\{r\}$, if the module is busy at t because it accepted a request r STUs ago, where r is an integer such that $1 \leq r \leq c-1$,

$=\{r'\}$, if the module is busy at t because it began a block transfer operation r′ STUs ago, where $1' \leq r' \leq (T-1)'$. □

A prime (′) is used only to distinguish a busy module which is involved in a block transfer operation from a busy module which is executing its cache cycle. Since numerical values of r can be equal to values of r′, the prime or absence of a prime is part of the state representation in addition to the numerical value. Definition 3.2.1.2 defines the line state, $\lambda(t)$, for this model also. The definitions which evaluate the next module state from a given module state are redefined as follows.

Definition 3.3.1.2 Given that the state of a module at t is ϕ, the next module state (at t+1) is

$=\{1\}$, if a hit request which addressed the module at t was accepted, or

$=\{1'\}$, if at time t, a request which addressed the module resulted in a miss and all the modules on the corresponding line were idle, i.e., the referenced module began a block transfer operation at t,

$=\phi$, otherwise; i.e., either no request addressed the module at t, or a request which addressed the module at t was rejected due to a line collision. □

Therefore, a module remains in the null state unless either it accepts a hit request at time t whereupon it will become busy and remain busy during the interval (t,t+c) or it begins a block transfer operation at time t whereupon it will become busy and remain busy during the interval (t,t+T). For a busy module, the next state can be evaluated by the following two definitions.

Definition 3.3.1.3 Given that the state of a module at time t is {r}, where r is an integer such that $1 \leq r \leq c-1$, the next module state is {r+1} if $r < c-1$ and ϕ if $r=c-1$. □

Once a module accepts a hit request, it goes through the module states {1}, {2},..., {c-1}, ϕ. As stated before, the maximum acceptance rate for a module is one accepted request per c STUs.

Definition 3.3.1.4 Given that the state of a module at time t is $\{r'\}$, where r' is an integer such that $1' \leq r' \leq (T-1)'$, the next module state is $\{(r+1)'\}$ if $r' < (T-1)'$ and ϕ if $r' = (T-1)'$. □

Clearly, once a module is involved in a block transfer operation it goes through the module states $\{1'\}, \{2'\}, \ldots, \{(T-1)'\}, \phi$. Again, determining the next line state is as straightforward as determining the next module state. The following definitions are needed to clarify the presentation.

Definition 3.3.1.5 If there exists $r' \in \lambda(t)$ such that $1' \leq r' \leq (T-1)'$, then $\lambda(t)$ is a busy line state, otherwise it is an idle line state. □

Observe that a busy line state contains one nonnull element with a prime ($'$). Once again, one element is sufficient to describe the busy line state even though there may be more than one module on the line actually involved in the block transfer operation. Note also that the sequence of busy line states can only start at line state ϕ. Since a miss can initiate a block transfer operation if and only if the line is at the null state, every accepted request will complete its cache cycle without being aborted. Hence every busy module is a blocking module.

Definition 3.3.1.6 An idle line state, $\lambda(t)$, is an acceptance state if $1 \in \lambda(t)$, otherwise it is a nonacceptance state. □

Since a hit or miss of a request is determined before it is accepted by the line or module, the checking state defined previously no longer exist in this case. In this model, a busy line state rejects all requests since the line is busy for block transfer operation. A nonacceptance state does not accept a request because either a request references a busy (blocking) module on the idle line, or no request references this idle line, or a request to this idle line is rejected due to a miss. There are two possible state transitions from each particular idle line state, $\lambda(t)$, except from the idle line state ϕ: one is to an acceptance state and the other is to a nonacceptance state. In addition to these two possible state transitions, there is one more possible state transition from the idle line state ϕ, that is to a busy line state. Now the transition probabilities can be obtained with the aid of the following lemma and corollaries.

Lemma 3.3.1.1 The probability of transition from an idle line state, $\lambda(t)$, to its successor acceptance line state is

$$P_a(\lambda) = \frac{m - |\lambda|}{m} hq,$$

where $|\lambda|$ is the cardinality of the line state $\lambda(t)$. □

The proof of this Lemma is similar to that for theorem 3.2.1.2. Note that every accepted request must be a hit request. Hence the probability of accepting a request at a particular idle line state is the probability

that a request references an idle module on the idle line, i.e., $q(m-|\lambda|)/m$, and the request is a hit.

Corollary 3.3.1.1.1 The probability of transition from a nonnull idle line state, $\lambda(t)$, to its successor nonacceptance line state is

$$P_n(\lambda) = 1 - P_a(\lambda) \qquad \square$$

Note that $P_a(\lambda)+P_n(\lambda)=1$ because there exist only two next line states: an acceptance state and a nonacceptance state.

Corollary 3.3.1.1.2 The probability of transition from line state ϕ to its successor busy line state is $P_b(\phi)=(1-h)q$ and the probability of transition from line state ϕ to its successor nonacceptance state is $P_n(\phi)=1-q$.

Proof: There are three possible next line states from the line state ϕ: the successor acceptance line state, the successor nonacceptance line state, and the busy line state. For line state ϕ, $P_a(\phi)=hq(m-0)/m=hq$. Thus $1-hq = P_n(\phi) + P_b(\phi)$. $P_n(\phi)=1-q$ is the probability that no request references the line at state ϕ and $P_b(\phi)=(1-h)q$ is the probability that a request which references the line at state ϕ results in a miss. $\square$

Similarly, since there is no successor acceptance state from a busy line state, the probability of transition from a busy line state , $\lambda(t)$,

to its successor busy line state (or to ϕ if $(t)=(T-1)'$) is one. Theorem 3.2.1.4 is also applicable in this case.

For given cycle characteristics (c,T), the line state diagram is readily constructed by using the above definitions and one-step transition probabilities. An example of the line state diagram for cycle characteristics $(c,T)=(3,10)$ and $m \geq c-1$ is shown in figure 3.3.1.1, where * indicates the acceptance states.

Theorem 3.3.1.2 The total number of distinct line states for cycle characteristics (c,T) is

$$\text{Total number of distinct line states} = \begin{cases} 2^{c-1} + T - 1, & \text{for } m \geq c-1 \\ \sum_{0 \leq i \leq m} \binom{c-1}{i} + T - 1, & \text{for } m < c-1 \end{cases}$$

Proof: For $m \geq c-1$, as before, the line states which contain nonnull element r such that $1 \leq r \leq c-1$ form a binary tree with tree height c-2. The total number of states contained in the binary tree is $2^{c-1}-1$. In addition, there are T-1 busy line states and one null line state. Hence the total number of distinct line states is $2^{c-1}+T-1$ for $m \geq c-1$. The number of busy line states is still T-1 for $m < c-1$. However, for $m < c-1$, the total number of distinct idle line states is $\sum_{0 \leq i \leq m} \binom{c-1}{i}$. □

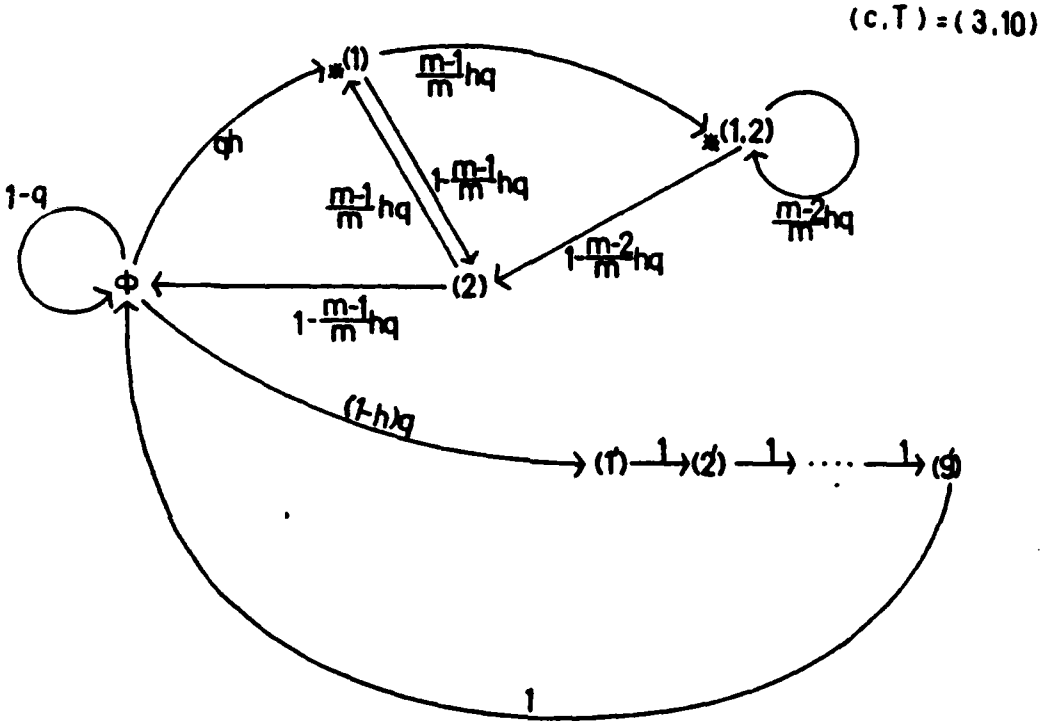


Figure 3.3.1.1 Line state diagram for shared cache with explicit lookup tables and cycle characteristics (c,T) = (3,10).

Theorem 3.3.1.3 In the L-M shared cache memory, the steady state probability, $P_{S\ell}(c,T,p)$, that a particular line at a particular STU accepts a hit request is

$$P_{S\ell}(c,T,p) = \sum_{\lambda \in \{\lambda \mid 1 \in \lambda\}} p_\lambda ,$$

where p_λ is the probability of being in line state λ. Furthermore,

$$P_{Ah}(c,T,p) = \ell P_{S\ell}(c,T,p)/hp$$

Proof: Since there are no aborted requests with explicit lookup tables under the selected scheduling strategy, a hit request is accepted, and therefore successful, at a particular line each time the line enters a state λ such that $1 \in \lambda$. The formula for $P_{S\ell}$ follows. Then $\ell P_{S\ell}$ is the number of accepted hit requests per STU over all lines and hp is the number of hit requests submitted per STU. The formula for P_{Ah} follows. □

Note that the line state diagram generated by the Markov model developed in this section is also an ergodic Markov Chain. Therefore, a unique equilibrium solution can be obtained.

The example of (c,T)=(3,10) can then be solved as follows.

Let p_i denote the probability of being in the line state i.

From figure 3.3.1.1, we can obtain the following six equations.

$$p_\phi = (1-q)p_\phi + \frac{m-mhq+hq}{m} p_2 + (1-h)qp_3 \qquad (1)$$

$$p_1 = qhp_\phi + \frac{m-1}{m} hqp_2 \tag{2}$$

$$p_{1,2} = \frac{m-1}{m} hqp_1 + \frac{m-2}{m} hqp_{1,2} \tag{3}$$

$$p_2 = \frac{m-mhq+hq}{m} p_1 + \frac{m-mhq+2hq}{m} p_{1,2} \tag{4}$$

$$p_1' = (1-h)qp_\phi \tag{5}$$

$$p_\phi + p_1 + p_2 + p_{1,2} + (T-1)p_1' = 1 \tag{6}$$

From equation (3),

$$\frac{m-mhq+2hq}{m} p_{1,2} = \frac{mhq-hq}{m} p_1$$

$$p_{1,2} = \frac{mhq-hq}{m-mhq+2hq} p_1 .$$

From equation (4),

$$p_2 = \frac{m-mhq+hq}{m} p_1 + \frac{mhq-hq}{m} p_1$$

$$= p_1 .$$

From equation (2),

$$p_1 = hqp_\phi + \frac{mhq-hq}{m} p_1$$

$$p_\phi = \frac{m-mhq+hq}{mhq} p_1 .$$

From equation (5),

$$p_1' = (1-h)qp_\phi$$

$$= \frac{(1-h)(m-mhq+hq)}{mh} p_1 .$$

Hence equation (6) becomes

$$\left[\frac{m-mhq+hq}{mhq} + 1 + 1 + \frac{mhq-hq}{m-mhq+2hq} + 9 \frac{(1-h)(m-mhq+hq)}{mh} \right] p_1 = 1.$$

Solving for P_1 yields

$$P_1 = \frac{Nhq(N - m\ell hq + 2\ell hq)Y_2}{Y_1Y_2 + 2NhqY_2 + Nhq9m\ell hq - \ell hq) + 9(1-h)qY_1Y_2}$$

where $Y_1 = (N - m\ell hq + \ell hq)$

and $Y_2 = (N - m\ell hq + 2\ell hq)$.

$$P_S\ell = P_1 + P_{1,2} = \frac{hN(1-P_1)(N+\ell hq)}{\Delta}.$$

Hence $$P_S = \frac{\ell hN(1-P_1)(N+\ell hq)}{\Delta}$$

and $$P_{Ah} = \frac{\ell N(1-P_1)(N+\ell hq)}{\Delta}.$$

where $\Delta = \ell N(N+\ell hq) + 9(1-h)\ell qY_1Y_2 + 2\ell^2 hq(N+\ell hq)$

Similarly, T can be left unspecified when constructing the line state diagram and a closed form solution is still obtained. For example, if the line state (9′) in figure 3.3.1.1 is replaced by ((T-1)′), then a line state diagram for (c,T)=(3,T) results and a closed form solutions for c=3 with general T can be obtained by going through the same computation process.

3.3.2 Probabilistic Model

In this section, the probabilistic approach is discussed. As stated

previously, a request may be blocked for one of four different reasons. These four types of blocking factors are in a different order in this model because of the different organization considered here. Since an explicit lookup table is associated with each line in this case, the line status will be checked after a hit or miss is determined for a request which references the line. Hence a cache request may be blocked due to

(1) Multiple access line collision (only if $p > 1$),

(2) Cache miss (only if $h < 1$),

(3) Busy line collision (only if $h < 1$ and $T > 0$), or

(4) busy module collision (only if $c > 1$).

Since no accepted request will be aborted, the busy line collision only rejects requests because the line is busy for a block transfer operation. Note also that the number of busy modules equals the number of blocking modules in this case. Let P_1, P_2, P_3 and P_4 be the probability of blocking of a request due to the above four events, respectively. Since these four events are mutually exclusive and independent of each other, theorem 3.2.2.1 and corollary 3.2.2.1.1 can still be applied.

Note that P_2 is equal to $(1-h)$ in this case and P_1 is still given by theorem 3.2.1.1. Only the probabilities of busy line collision, P_3, and busy module collision, P_4, have to be rederived.

Lemma 3.3.2.1 The probability of a request referencing a busy line is

$$P_3 = \frac{q(T-1)(1-h)E(\phi)}{\ell},$$

where $E(\phi)$ is the expected number of lines in line state ϕ and q is given in lemma 3.2.1.1.1.

Proof: Let p_ϕ be the probability that a particular line is in line state ϕ. Note that an idle line state will make a transition to a busy line state if and only if a miss occurs on a line at state ϕ. Hence the probability that a particular line changes from idle to busy is $(1-h)qP_\phi$. Since there are ℓ lines in the system, the expected number of idle lines which become busy lines per STU is $(1-h)q\ell P_\phi$. Once a line becomes busy, it remains busy and rejects all newly arriving requests for the following T-1 time units. Therefore, the expected number of total busy lines, E(BL), seen by an arriving request is $E(BL) = (T-1)(1-h)q\ell P_\phi$ Since $E(\phi) = \ell p_\phi$, then $E(BL) = (T-1)(1-h)qE(\phi)$. By definition, $P_3 = E(BL)/\ell = (T-1)(1-h)qE(\phi)/\ell$. □

Since there are no aborted requests in this model, one additional busy (blocking) module will be caused in the system by each accepted hit request. Thus, the computation of the probability of referencing a busy module on an idle line, P_4, is straightforward.

Lemma 3.3.2.2 The probability of a hit request referencing a busy module on an idle line is

$$P_4 = \frac{(c-1)pP_S}{N(1-P_3)}$$

Proof: Since every busy module is a blocking module in this case, by lemma 3.2.2.2, P_4=E(BM/IL)/E(M/IL). The expected number of idle lines is given by corollary 3.2.2.1.2, i.e., $\ell_{idle}=\ell(1-P_3)$ in this case. Hence, P_4 = E(BM/IL)/N(1-P_3). Obviously, the expected number of busy modules on idle lines, E(BM/IL), is the total number of accepted hit requests during the last (c-1) time units. Since there are p simultaneous requests made by a parallel-pipelined processor of order (s,p) per STU, E(BM/IL) is then equal to $(c-1)pP_S$. □

The only unknown needed to be solved for in order to obtain the probability of acceptance of a hit request is E(ϕ). The following lemma aids in the evaluation of E(ϕ).

Lemma 3.3.2.3 In the steady state, the total number of successful requests per STU for a system with cycle characteristics (c,T) is

$$pP_S = \sum_{\lambda \in \{\lambda | 1 \in \lambda\}} E(\lambda)$$ □

This is obvious since every accepted request is a successful request.

The probability of acceptance, $P_{Ah}(c,T,p)$, of a hit request for the cycle characteristics (3,10) can now be solved for as follows.

$$P_1 = 1-[1-(1-1/\ell)^p]\ell/p$$

$$P_2 = 1-h$$

$$P_3 = (T-1)(1-h)qp_\phi$$

$$P_4 = \frac{(c-1)pP_S}{N(1-P_3)} = \frac{2pP_S}{N(1-P_3)}$$

From figure 3.3.1.1. and lemma 3.3.2.3,

$$E(1) + E(1,2) = pP_S$$

and

$$E(1,2) = \frac{m-1}{m} hqE(1) + \frac{m-2}{m} hqE(1,2).$$

Solving for E(1,2):

$$E(1,2) = \frac{mhq-hq}{m-mhq+2hq} E(1)$$

Substituting for E(1,2) in equation (1) and simplifiying,

$$E(1) = \frac{m-mhq+2hq}{m+hq} pP_S.$$

From figure 3.3.1.1, E(2) = E(1) and

$$E(1) = \frac{m-1}{m} hqE(2) + hqE(\phi)$$

$$= hqE(\phi)+ \frac{m-1}{m} hqE(1)$$

Solving for $E(\phi)$,

$$E(\phi) = \frac{m-mhq+hq}{mhq} E(1)$$

$$= \frac{(m-mhq+hq)(m-mhq+2hq)}{mhq(m+hq)} pP_S.$$

Hence, $$P_\phi = \frac{(m-mhq+hq)(m-mhq+2hq)}{Nq(m+hq)} pP_{Ah}.$$

Substituting for p_ϕ and T in P_3

$$P_3 = (T-1)(1-h)qp_\phi = 9(1-h)\frac{(m-mhq+hq)(m-mhq+2hq)pP_{Ah}}{N(m+hq)}.$$

Substituting for P_3 and P_4 in

$$P_{Ah} = (1-P_1)(1-P_3)(1-P_4)$$

and solving for P_{Ah} yields

$$P_{Ah} = \frac{\ell N(1-P_1)(N+\ell hq)}{\ell N(N+\ell hq)+9(1-h)\ell qY_1Y_2+2\ell^2 hq(N+\ell hq)}$$

where $Y_1 = (N-m\ell hq+\ell hq)$ and $Y_2 = (N-m\ell hq+2\ell hq)$

Here P_{Ah} is identical to the result obtained in last section.

In order to evaluate the performance, CPU utilization, a knowledge of the penalties caused by a cache miss is needed. The following lemma helps to evaluate the miss penalty due to rejection only.

Lemma 3.3.2.4 *The probability of acceptance for a miss request attempting to initiate a block transfer operation is*

$$P_{Am}(c,T,p) = (1-P_1)p_\phi ,$$

where p_ϕ is the probability of being in line state ϕ.

Proof: A miss request is accepted, i.e. a block transfer operation is initiated, if and only if this miss request passes through the multiple access line collision, probability $(1-P_1)$, and references a line in state ϕ, probability p_ϕ. Since these two events are mutually exclusive and independent of each other, the lemma follows. □

Theorem 3.3.2.1 The CPU utilization, Cu, for a shared cache memory with explicit lookup tables and the cycle characteristics (c,T) is

$$Cu = \frac{1}{\frac{h}{P_{Ah}} + (1-h)\left[\frac{1}{P_{Am}} + T''\right]}$$

where $T'' = \lceil T/s \rceil$.

Proof: From the derivation of theorem 3.2.2.2, it is easy to show that given a probability of acceptance, P_A, for a request, the penalty, i.e., null passes, caused by the access conflict is $(1/P_A)-1$. However, in the case of the shared cache memory with explicit lookup tables, P_{Ah} is the probability of acceptance for a hit request. Hence the hit requests will suffer a penalty of $(1/P_{Ah})-1$. Similarly, from lemma 3.3.2.4, a cache miss has to wait for $(1/P_{Am})-1$ null passes in order to initiate the block transfer operation. Hence the total penalty caused by a miss request is $(1/P_{Am})-1+T''$. The total number of passes a random request must take is

$$\left(\frac{1}{P_{Ah}} - 1\right)h + \left[\frac{1}{P_{Am}} - 1 + T''\right](1-h) + 1$$

$$= \frac{h}{P_{Ah}} + (1-h)\left[\frac{1}{P_{Am}} + T''\right].$$ □

For the cases c=1, 2, and 3, a summary of the probabilities of acceptance for hit and miss requests are listed in Appendix B. These calculations require reference to the corresponding Markov state diagrams, at least for evaluation of $E(\phi)$, and no useful analytic bound on performance has been found for the explicit lookup table models.

3.3.3 Dynamic Hit Ratio

Due to the complexity of evaluating the hit ratio function, the hit ratio is left unevaluated and treated as a specified parameter in our analytic models. The hit ratios for various parameters can be obtained from simulation. The hit ratio (miss ratio) obtained from simulation for a given program and a particular set of parameters is called static hit ratio (static miss ratio).

For shared cache with explicit lookup tables, a miss request can be accepted, i.e. a block transfer operation is initiated, by a line only if that line is in the empty state. Therefore, a miss request is more likely to be rejected by this cache memory system than a hit request. Due to this biased rejection for miss and hit requests, more miss requests will be resubmitted than hit requests. Then a lower hit ratio, called dynamic hit ratio, rather than the static hit ratio is seen by the cache memory system. The dynamic miss ratio is similarly defined.

In sections 3.3.1 and 3.3.2, the derivation of the probabilities of acceptance for miss and hit requests, namely P_{Ah} and P_{Am}, involves an

independent parameter, i.e. the given hit ratio h. This hit ratio h should be the dynamic hit ratio actually seen by the cache memory system for the reason mentioned above. Therefore, the parameter h which appears in the transition probabilities and the probabilities of acceptance for explicit lookup table model is the dynamic hit ratio, instead of the static hit ratio. However, the hit ratio h which appears in the CPU utilization formula given by theorem 3.3.2.1 is the static hit ratio since the number of misses which actually cause a block transfer operation is assumed unchanged. Note that miss and hit requests are equally likely to be rejected for the implicit lookup table model. Therefore, the dynamic hit ratio equals the static hit ratio for implicit lookup table model.

Let h_s and h_d denote the static hit ratio and dynamic hit ratio, respectively. The relationships between static hit ratio and dynamic hit ratio for the explicit lookup table model can be derived as follows. Since a line state, λ, with $1 \in \lambda$ accepted a hit request one STU ago and a line state, λ, with $1' \in \lambda$ accepted a miss request one STU ago, the static hit ratio can then be expressed as

$$h_s = \frac{\sum_{\lambda \in \{\lambda |\ 1 \in \lambda\}} P_\lambda}{\sum_{\lambda \in \{\lambda |\ 1 \in \lambda \text{ or } 1' \in \lambda\}} P_\lambda},$$

where p_λ is the probability of being in line state λ.

By substituting h_d for h in the transition probabilities developed

in section 3.3.1, the relationships between h_s and h_d can be solved from line state diagram for a specified cache cycle, c. For a given c, h_d can be expressed in terms of h_s. The performance prediction for the explicit lookup table model can be computed by replacing h with h_d in P_{Ah} and P_{Am} and then plugging P_{Ah}, P_{Am} and $h=h_s$ in the CPU utilization formula.

Since this relationship is dependent on c, no general solution of dynamic hit ratio for arbitrary c is obtained. A summary of the relationships between static hit ratio and dynamic hit ratio for the cases c=1, 2, and 3 is listed in Appendix B.

3.4 Private Cache Memories

All the previous sections in this Chapter deal with the performance analysis of shared cache memory systems. In order to compare the performance difference between a shared cache memory and private cache memories, a probabilistic model for multiprocessor system with private cache memories is discussed in this section.

Figure 1.5.1 shows the organization of a multiprocessor system with private cache memories. Since each stream has its own private cache and since there is no overlap within a stream, there is no cache access conflict. Furthermore, a request which results in a cache miss will immediately cause the cache controller to generate a request, called a main memory request, to the main memory for fetching the new block. However, since the main memory is shared by all the processors in the

system, a main memory request may be rejected due to access conflict. Therefore, the system performance is dependent not only on the hit ratio but also on the access conflict at main memory. As before, the hit ratio will be left unevaluated and considered as a parameter which can be obtained from simulation. The analytical model here is oriented toward developing the probability of acceptance, P_{AM}, of a given main memory request for the private cache system with a shared main memory.

In addition to the assumptions made in section 3.1, more explanation is necessary to clear up some possible ambiguities in this system. Note that the p parallel processors in this case are nonpipelined processors. Therefore, one instruction (processor) cycle time, instead of one STU, is considered as the basic time unit. The processor request rate, ψ, is assumed to be one as before. Thus each processor makes one cache request every instruction cycle. Since the processor request rate, ψ, equals one, the miss ratio, 1-h, becomes the request rate for main memory requests from each processor. The occurrence of a cache miss is independent of previous misses in the same private cache and in the other private caches. Also it is independent of all the other simultaneous cache misses. From the main memory point of view, since each processor executes its own independent stream, the addresses of the main memory requests are randomly distributed. For analytical simplicity, it is assumed that the addresses of the main memory requests are independent and uniformly distributed among the M main memory modules. The independence and randomness assumption for the main memory reference patterns allows rejected requests to be discarded in the model. In

practice, those rejected requests will be resubmitted the very next cycle. The effects of the resubmitted requests will be tested in the simulation model discussed in chapter 4. Note that if the main memory modules are interleaved by low-order bits, the crossbar is then switched very often during the block transfer operation. Thus, a main memory interleaved by high-order bits or interleaved by blocks is assumed.

Once a main memory module accepts a request, the main memory module, the associated line, and the associated connection path in the crossbar are busy for T" time units, where T" is the total number of instruction cycles needed to complete a block transfer operation. Therefore, a request to the main memory may be rejected for either of the following two independent and mutually exclusive events.

(1) Multiple access line collision (only if $p > 1$), or
(2) Busy (main memory) module collision (only if $h < 1$ and $T" > 0$).

Let P_I and P_{II} be the probabilities of rejection of a main memory request due to (1) and (2), respectively. The probability of a main memory request being accepted, P_{AM}, is then $(1-P_I)(1-P_{II})$. The following theorems are developed to evaluate P_{AM}.

Lemma 3.4.1 The probability of a main memory request being rejected due to multiple access line collision is

$$P_I = 1 - \frac{M}{(1-h)p}\left[1 - \left(1 - \frac{1-h}{M}\right)^p\right],$$

where (1-h) is the main memory request rate and M is the number of main memory modules.

Proof: Since the request rate of each processor is (1-h), there are a total of (1-h)p requests issued every instruction cycle. Obviously a particular memory module in a particular cycle will be referenced by a request with probability (1-h)/M. Similar to the derivation of theorem 3.2.1.1, the memory bandwidth is $M\{1-[1-(1-h)/M)]^p\}$. The probability of acceptance, $1-P_I$= (memory bandwidth)/(expected number of requests per cycle). □

Theorem 3.4.1 The probability of acceptance, P_{AM}, of a main memory request for a multiprocessor system with private cache memories is

$$P_{AM} = \frac{\frac{M}{(1-h)p}\left[1 - \left(1 - \frac{1-h}{M}\right)^p\right]}{1 + \left[1 - \left(1 - \frac{1-h}{M}\right)^p\right]\left[T''-1\right]}$$

Proof: There are an average of (1-h)p main memory requests issued every processor cycle. Of these, a total of $(1-h)p(1-P_I)$ requests pass through the multiple access line collision. A particular main memory module is referenced by one of those requests with probability

$(1-h)p(1-P_I)/M$. Therefore, the request rate seen by each main memory module is $(1-h)p(1-P_I)/M$. Once a main memory module accepts a request, it will be busy for T" time units. Figure 3.4.1.1 illustrates the Markov state diagram for a main memory module. The definition of the module state is similar to Definition 3.3.1.1. Note that only an idle module can accept a request. The module states of nonnull elements represent the busy module states. Since ϕ is the only idle module state, the probability of being in state ϕ, p_ϕ, is $1-P_{II}$. From this state diagram, p_ϕ can easily be found and is given as follows:

$$p_\phi = \frac{1}{1 + \frac{(1-h)p(1-p_I)}{M}\ (T"-1)},$$

Therefore, $P_{AM} = (1 - P_I)(1 - P_{II})$.

$$= \frac{\frac{M}{(1-h)p}\ [\ 1 - (\ 1 - \frac{1-h}{M}\ )^p\ ]}{1 + [\ 1 - (\ 1 - \frac{1-h}{M}\ )^p\ ][\ T" - 1]} \qquad \square$$

Note that if $\alpha = (1-h)p(1-P_I)/M$, then $p_\phi = 1/[1+\alpha(T"-1)]$ is the probability of acceptance of a request for a single resource, where α is the resource request rate and T" is the resource cycle. Emer [60] has derived this result.

One assumption used to derive theorem 3.4.1 is that the main memory request rate, i.e. static request rate, is (1-h). However, this rate

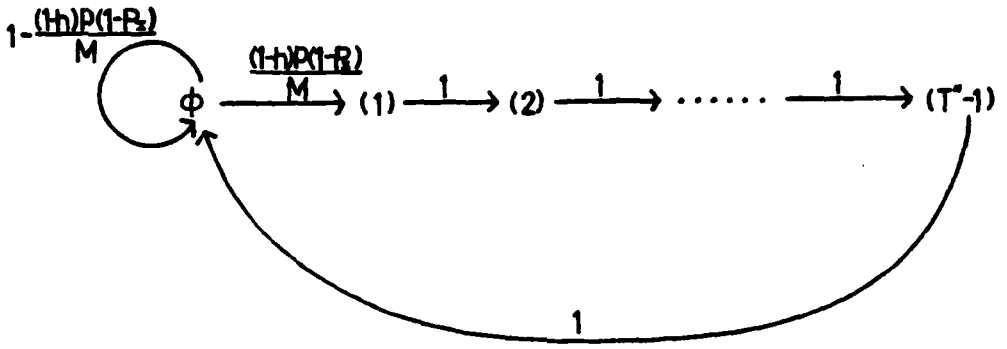


Figure 3.4.1.1 Memory line state diagram for multiprocessor with private cache systems.

requires adjustment because a main memory request will be resubmitted if it is rejected. Furthermore, once a main memory module accepts a request made by a particular cache controller, this cache controller would not make any new request during the following T" time units. With these two effects, the actual request rate, i.e. dynamic request rate, seen by the main memory may be altered. The following theorem gives the dynamic request rate seen by the main memory by considering the request rate changes due to rejection and acceptance.

Theorem 3.4.2 For a given static request rate, (1-h), the dynamic request rate seen by the main memory is

$$\alpha = \frac{1}{1 + T''P_{AM} + \left(\frac{1}{1-h}\right)P_{AM}}.$$

Proof: Assume that the total number of instruction cycles needed to execute a job is R+R′ if there is no main memory access conflict and the main memory cycle is less than one instruction cycle, where R of the R+R′ processor cycles involve requests. Then the request rate is R/(R+R′)=(1-h). However, a request will be resubmitted the very next cycle if it is rejected due to memory access conflict. On the average, a request takes $1/P_{AM}$ instruction cycles in order to be accepted, where P_{AM} is the probability of acceptance of a request. During each null pass, the same blocked request is resubmitted. Therefore, R requests

will be extended to a total of R/P_{AM} requests. Once a request made by a particular processor is accepted, no new request will be made by this processor for the following T" time units if the memory cycle time is T". Since every request will eventually be accepted, the total number of instruction cycles in which there is no request is $R'+RT"$. Hence the dynamic request rate seen by the memory is

$$\alpha = \frac{\frac{R}{P_{AM}}}{\frac{R}{P_{AM}} + R' + RT"}$$

Since $1-h = R/(R+R')$, then $\alpha = \dfrac{1}{1 + T"P_{AM} + (\frac{1}{1-h} - 1)P_{AM}}$. □

Note that the dynamic request rate, α, can be larger than or smaller than the static request rate, $(1-h)$. If $T" = 0$, then $\alpha = 1/\{1+[(1/1-h) - 1]P_{AM}\}$, which is the result given in [60]. In this case, since the block transfer time is zero, the above theorem reduces to the case in which the system contains processor and main memory with processor request rate 1-h and main memory cycle time one. Note α is always larger than 1-h for $T"=0$ because every rejected request will be resubmitted which increases the request rate. At another extreme, if h=0, then $\alpha = 1/(1+T"P_{AM})$. In this case, the theorem reduces to the case in which the unadjusted processor request rate is one and main memory cycle time is T". Now α is

always smaller than one, since once a request made by a particular processor is accepted, this processor will not make any new request until the accepted request finishes its cycle.

This actual request rate tries to correct the assumption made in theorem 3.4.1. Note that α is a complex function of P_{AM}. Thus this equation is most easily solved by iteration. By combining theorem 3.4.1 and theorem 3.4.2, the corrected solution for P_{AM} can be obtained by the following two iterative equations provided an initial condition for α_i is given:

$$P_{AM_{i+1}} = \frac{\frac{M}{\alpha_i P}\left[\,1 + \left(1 - \frac{\alpha_i}{M}\right)^P\,\right]}{1 + \left[\,1 - \left(1 - \frac{\alpha_i}{M}\right)^P\,\right]\left[\,T'' - 1\,\right]}$$

$$\alpha_{i+1} = \frac{1}{1 + T''P_{AM_{i+1}} + \left(\frac{1}{1-h} - 1\right)P_{AM_{i+1}}},$$

given $\alpha_0 = 1 - h.$

Theorem 3.4.4 The CPU utilization, Cu, for a multiprocessor system with private cache nmemories is

$$Cu = \frac{1}{1 + (1-h)\left[\frac{1}{P_{AM}} - 1 + T''\right]}$$

Proof: This is obvious since the penalty for a miss is $(1/P_{AM})-1+T''$.

For the private cache memories, there is no access conflict at the cache level. Therefore, there is no penalty for a hit cache request. □

3.5 Concluding Remarks

The analytic models for two distinct cache organizations, namely shared cache with an implicit lookup table and shared cache with explicit lookup tables, have been developed in this chapter. A Markov approach and a probabilistic approach have been presented for both models. Due to the complexity of evaluating the hit ratio function, the hit ratio is left unevaluated and treated as a specified parameter in our analytic models.

For shared cache with an implicit lookup table, a block transfer operation is initiated on a line as soon as a miss is detected on that line. Any incompletely served request on the same line when a miss occurs is simply aborted. Since miss and hit requests are equally likely to be accepted, program static hit ratio is equal to the dynamic hit ratio in this case.

However, for shared cache with explicit lookup tables, a higher priority of acceptance is assigned to hit requests. A block transfer operation can be initiated on a line only if that line is in the empty state. In this case, a miss request is more likely to be rejected than a hit request. Due to the biased rejection for miss and hit requests, the dynamic hit ratio seen by cache memory system is different from the

program's static hit ratio. We have shown the relationship between static hit ratio and dynamic hit ratio.

Since the line state space increases exponentially with cache cycle, c, no general solution for performance has been obtained for arbitrary c. However, we have derived upper and lower performance bounds for the implicit lookup table model, but not for explicit lookup table model.

A probabilistic model for private cache systems has also been developed.

In chapter 4, the hit ratio function is evaluated for various parameters by simulation. The simulator is driven by real program traces. In addition to evaluating the hit ratio function, the inaccuracy caused by the assumptions about program referencing patterns for our analytic models is also evaluated by comparing the analytic predictions with simulation results. Furthermore, some dynamic space sharing behavior is observed and the effects of access conflict on performance are discussed for a wide variety of parameter values.

CHAPTER 4

ANALYSIS OF RESULTS

4.1 Introduction

Due to the complexity of the effects of various parameters on hit ratio, hit ratio was unevaluated and considered as a given parameter in analytic models. In this chapter, the hit ratio function is investigated for a range of parameters and several different workloads by simulation experiments. In the previous chapter, it was assumed that blocked requests were discarded so that the independent and random request assumption could be justified in the analytical models. The analyses in last chapter were also based on the assumption that the cache hit ratio is independent of cache access conflict. In this chapter, these assumptions about program behavior will be verified by comparing the analytical predictions with simulation results. Therefore, the purposes of simulation experiments are to study the hit ratio function and to validate the analytic models. In addition to the discussion about the simulation results, the effects of varying the ℓ , N, p, c, T, τ and h parameters will be discussed based on analytic predictions. The performance prediction for processors with load through capability is illustrated by a direct extension of the analytic solutions developed in the previous chapter. Furthermore, the performance comparisons between

shared cache and private cache for some sets of parameters are given. For convenience, from now on, the analytic models of the shared cache memory systems with implicit lookup table and explicit lookup tables will be called model A and model B respectively.

Trace-driven simulators written in SIMULA [61], for model A and model B, have been developed. Four real program traces are used in the simulation study to generate address sequences for cache memory requests. They are GAUSS, EIGEN, ECOBOL, and CCOBOL. The first program, GAUSS, performs Gaussian elimination on a 20x20 matrix to solve a set of simultaneous linear equations. The second program, EIGEN, determines the eigenvalues of a 14x14 matrix. Both GAUSS and EIGEN were written in FORTRAN. ECOBOL is the trace of an execution of a COBOL program and CCOBOL is the trace of a compilation of a COBOL program. All the traces were collected by running these programs on an IBM/360 system.

Although the multiple-stream systems investigated in this thesis are MIMD computer systems as introduced in chapter 1, they can be further classified into the following four different operating environments: Independent Instruction - Independent Data (IIID), Shared Instruction - Independent Data (SIID), Independent Instruction - Shared Data (IISD), and Shared Instruction - Shared Data (SISD). A MIMD computer system operates in the IIID environment if there is no shared instructions and data among streams. This is the usual situation when each processor executes its own program and data. However, if all processors execute the same program, but each operates on a different data set, the SIID operating environment pertains.

In this research, IISD and SISD operating environments will not be investigated in the simulation studies because no program trace corresponding to these environments is available. However, IIID and SIID operating environments can be simulated by the four available program traces and their effects on hit ratio will be studied. For the IIID operating environment, shared cache may result in higher performance than private cache due to dynamic space sharing. In addition to the dynamic space sharing, shared blocks, i.e. blocks containing shared information, may further improve the performance for shared-cache systems operating in the SIID environment. To simulate a p-processor system, p program traces should be used. In our simulation study, each program trace is evenly divided into four trace sections and thereby up to 16 processors can be simulated simultaneously.

The IIID operating environment can then be simulated by assigning each processor its own unique trace section and an associated offset constant. The effective addresses requested by each processor are generated by adding its offset constant to each address in the associated trace section. The offset constants are chosen such that the effective address spaces of both instructions and data among the p trace sections are disjoint.

The SIID operating environment can be simulated by assigning each processor a unique trace section of a single program trace and an associated offset constant. For each processor, its offset constant is added to the data addresses in the associated trace section to generate the effective addresses. The offset constants in this case are chosen

such that the effective data address spaces of the p processors are disjoint. This offsetting technique in trace-driven simulation has been used by some previous authors [40] to investigate the memory interference problem in multiprocessor computer systems.

The effects of different operating environments on the hit ratio will be discussed in section 4.3. However, the IIID operating environment is used in the remainder of this chapter. In section 4.4, performance comparisons between shared cache and private cache for a range of parameters are carried out to validate the analytic models.

Due to the fact that the architecture of the IBM/360 uses various instruction lengths and various length data representations, the traced addresses are not all at the word boundaries. In the simulation study, it is simply assumed that each word contains four bytes and both cache capacity and block sizes are integral multiples of the word size. A cache request is generated by adding the offset constant to the traced address and then ignoring the two least significant bits. For example, a cache memory with block size 8 means each block contains 8 words, or 32 bytes.

A high rate of cache misses usually happens during the initial period of the execution of a particular program because of an initially empty cache. As the cache begins to fill, the initial high rate of cache misses drops rapidly and soon reaches the value for a full cache [46]. Cold-start miss ratios are miss ratios that are measured with an initially empty cache memory [62]. Warm-start miss ratios are miss

ratios that are measured with a cache which is full with the blocks associated with the process being executed. Cold-start miss ratios are useful in studying certain aspects of multiprogramming performance because cache miss ratios are affected by task switching. On the other hand, warm-start miss ratios should be measured if behavior of a program running uninterrupted for indefinitely long periods of time is being studied. Hence, all miss ratios referred to in this chapter are the warm-start miss ratios.

Let the time period between the beginning of a simulation run with an initially empty cache and the time instant to measure the warm-start miss ratio be the initial period. Usually this initial period is the time period needed to fill the cache memory. However, the cache memory may never be filled in our experiments. Since the set associative mapping mechanism is used, the address space of a program may not be mapped onto all the cache blocks. In this simulation study, the initial period is determined heuristically by experiment. It was found that the time period of the first five thousand distinct requests is sufficiently long for a 1K cache memory to avoid the effect of the initial condition. Therefore, the time period of the first five thousand distinct requests is used as the initial period for a 1K cache memory in our simulation experiments. Since the initial period is a function of the total cache capacity, the initial period is proportionately increased when the cache capacity is increased. For example, the time period of the first twenty thousand distinct requests is used as the initial period for a 4K cache memory.

In addition to the initial conditions, the total length of the simulation run, including the initial period, is important for properly interpreting the measured data. The simulated system may not reach the steady state if the simulation is terminated too early. However, it is expensive for a long simulation run. In our experiments, the simulation was performed for a certain number of instruction cycles, i.e. number of cache requests, because of simulation costs. This termination time is also heuristically determined by experiment. For each trace section, it was found that there are no significant changes in the measured performance data after a total of ten thousand distinct requests, including the initial period, has been made. Therefore, the termination time of the simulation running under a single stream environment is the time when the ten thousand-th distinct request is made. Note that a parallel-pipelined processor of order (s,p) can execute sp distinct instruction streams concurrently. Those sp requests made within one instruction cycle are interleaved among the streams so that each comes from a distinct stream. Therefore, the termination time for the multiple stream cases is a function of the total number of streams, sp, in the system. In this study, the termination time is proportionately increased when the number of streams is increased. For example, to simulate the case of four streams sharing a 1K cache memory, the initial period is set to the time when the five thousand-th distinct request is made and the termination time is set to the time when the forty thousand-th distinct request is made.

General definitions of hit ratio and miss ratio were introduced in

chapter 1. This definition was applied in chapter 3 under the independent request assumption of the analytical models. However, in the simulation experiments, the blocked requests are resubmitted, instead of discarded, until they are satisfied. The hit ratio is then a function of the number of resubmitted requests if the previous definition of hit ratio, i.e. the fraction of all cache memory requests resulting in a hit, is used in analyzing the simulation data. In order to exclude the effect of those resubmitted requests on the hit ratio, the hit ratio in the simulation experiments is defined as the fraction of all distinct cache requests (not including the resubmitted requests) resulting in a hit. The miss ratio is similarly defined. The CPU utilization in the simulation experiments is defined as the fraction of total cache requests (including the resubmitted requests) resulting in a hit. This is equivalent to the fraction of time that CPU is busy doing useful work.

In summary, four different operating environments and program traces have been introduced in this section. Cold-start and warm-start measurements were also discussed. The initial period and simulation termination time have been determined. In the following discussion, the write-through updating scheme is assumed for shared-cache systems and the write-back updating scheme is assumed for private-cache systems, except when stated otherwise. Except for section 4.3, the IIID operating environment is assumed in the rest of this chapter. Note that it is impractical to show the combined effects of all parameters pictorially on a two-dimensional graph. A simplification, which is adopted here, studies the effect of each variable on performance, independently. Note

also that the hit ratio (or miss ratio) in the remainder of this chapter means the static hit ratio (or static miss ratio), except when stated otherwise.

4.2 The Effects of Block Size, Set Size and Total Cache Capacity on Miss Ratios

For a complicated system with many parameters, the effects of a particular parameter on the system performance can be investigated by varying this parameter while holding the other parameters constant. In this research, the effects of the hit ratio and the cache access conflict on system performance are treated as two different issues and studied separately. In the analytical models, they are separated by the assumption of independence between them. However, in the simulation experiments, the effects of cache access conflict on system performance can be eliminated by setting c=T=0 and p=1 while the effects of hit ratio on the system performance are investigated. Note that for p=1, the number of streams in the system can be specified by s, the number of segments per pipelined proessor. Thus the effects of various parameters on the hit ratio can be studied by setting c=T=0 and p=1.

In multiple-stream shared-cache computer systems, the hit ratio is a complicated function of the number of streams, the block size, the set size, the total cache capacity, and the program characteristics. The effects of program characteristics on the hit ratio are discussed in the following section. In this section, the effects of varying the set size,

the block size, and the cache capacity on the hit ratio for a fixed number of streams are investigated. Note that the number of segments, s, in each pipelined processor has no effect on the cache access conflict [41]. The effect of the number of streams, sp, on the hit ratio can be determined by varying s with p=1 because no cache access conflict, i.e. c=T=0 and p=1, is considered. In the following discussions, s is simply assumed to be four, except when stated otherwise.

Figure 4.2.1 shows the miss ratio vs. cache capacity provided the set size and block size are both fixed at 8. In this case, the first section of each program trace is used to simulate the IIID operating environment. The cache capacity is the total cache size measured in words. Therefore, each private cache has cache size of a quarter of the specific cache capacity. The miss ratio for the private-cache system is the value averaged over the four streams. For this workload, figure 4.2.1 shows that shared cache always performs better than private cache. This miss ratio improvement of shared cache may be caused by dynamic space sharing and/or the write through policy used for shared cache. The effects of dynamic space sharing and write policies on miss ratio will be investigated separately in section 4.3. This figure shows that the largest difference in miss ratio between shared cache and private cache happens at a cache capacity of 1K. This difference becomes smaller as the cache capacity either increases or decreases from 1K words. For small cache capacity, the difference of miss ratios is small because all cache blocks in both shared cache and private cache become saturated. For large cache capacity, this difference is small because the cache

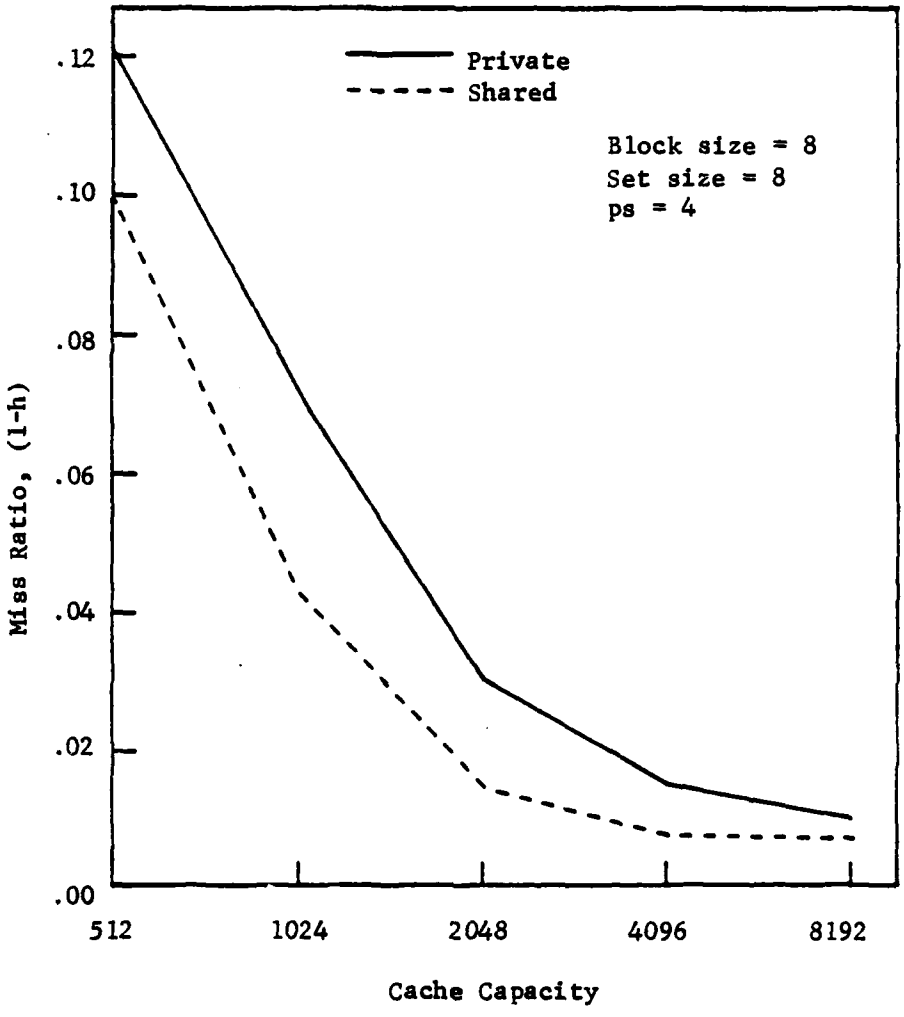


Figure 4.2.1 Effect of cache capacity on miss ratio.

memories may contain most of the information needed for execution and the cache hit ratios approach one for both shared cache and private cache. Similar observations have been pointed out by Coffman and Ryan [29].

The impact of block size is shown in figure 4.2.2. The label at the right side of each curve indicates the cache capacity. As can be seen in most cases, for a fixed cache capacity, the miss ratio tends first to decrease as the block size increases and then increases after a minimum is reached. Especially in smaller caches, the miss ratio significantly decreases as the block size increases. This happens because a smaller cache depends more on the prefetching, i.e. block fetching, effect for its performance. Since small caches may not be able to keep program loops, the miss ratio improvement for large block sizes is primarily due to block fetching which matches program sequentiality and spatial locality. For a fixed cache capacity, the increase in miss ratio results from the blocks becoming so large that too few blocks are contained in the cache. If the cache capacity increases to always contain the same number of blocks, the miss ratio will continue to decrease as block size is increased (with no adjustment in T). As shown in figure 4.2.2, for example, the block size corresponding to minimum miss ratio for private cache increases from 4 to 8 as the cache capacity increases from 1024 to 2048. These observations are also consisent with the results of previous studies [45,46].

Figure 4.2.2 also shows that the shared cache performs significantly better than the private cache in this case. More specifically, the relative miss ratios of shared cache with respect to private cache with

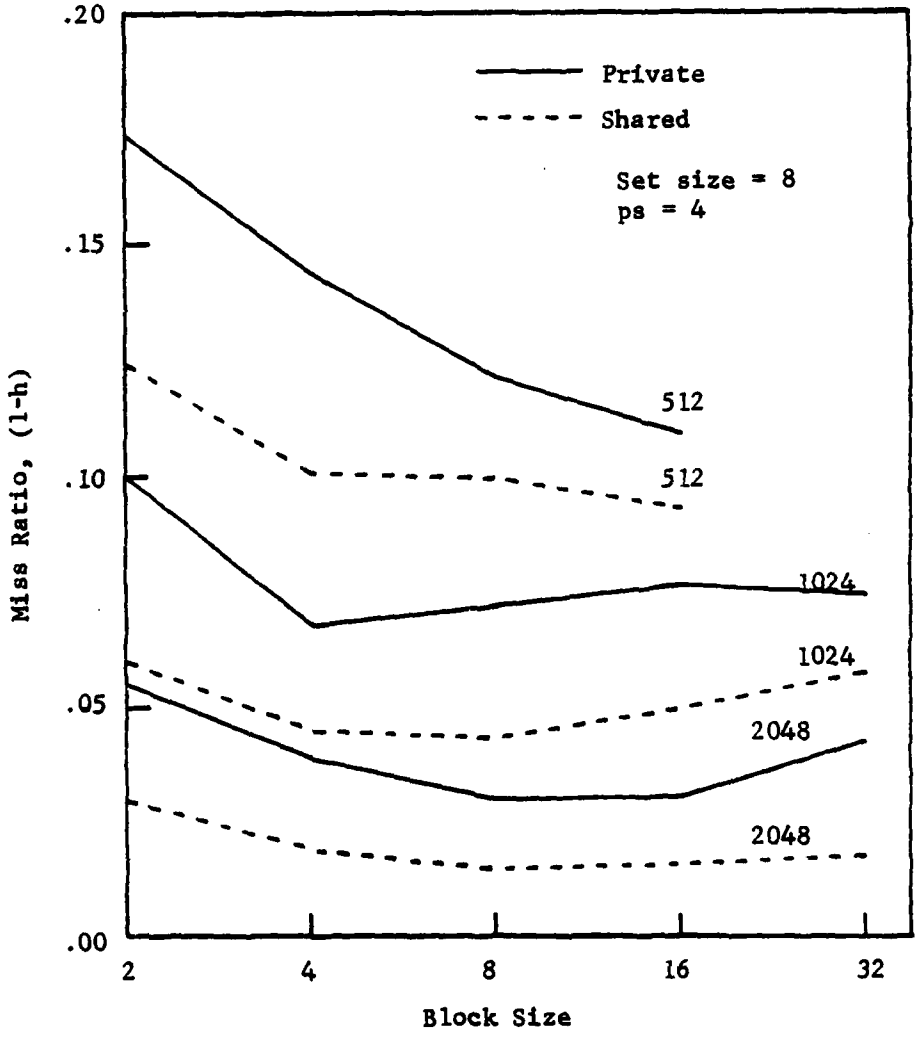


Figure 4.2.2 Effect of block size on miss ratio.

various block sizes are 0.426 to 0.541 for a 2 K cache, 0.595 to 0.777 for a 1 K cache, and 0.714 to 0.852 for a 256-word cache. The minimum miss ratios for shared cache with 512, 1024 and 2048 cache capacities are 0.093, 0.043 and 0.015, respectively. However, the minimum miss ratios for private cache with 512, 1024 and 2048 cache capacities are 0.109, 0.067 and 0.03, respectively.

As illustrated in theorem 3.2.2.2 and theorem 3.3.2.1, the performance is not dependent on miss ratio only, but also on the block transfer time T" measured in processor cycles. The block transfer time is a function of the block size and the bandwidth of the main memory. Larger blocks imply the need for longer block transfer time for a fixed main memory bandwidth. Hence, the block size corresponding to the minimum value of miss ratio for a given cache capacity may not result in optimum performance. To optimize the performance, the block transfer time T", and miss ratio (1-h) have to be minimized. The tradeoffs between miss ratio and block transfer time will be discussed in more detail in section 4.10.

Figure 4.2.3 depicts the effect of set size on miss ratio with fixed cache capacity and block size. The miss ratio decreases as the set size increases. The largest improvement in miss ratio occurs in going from set size one to set size two for both shared cache and private cache. The curve for private cache becomes flat as long as the set size is larger than four. The previous studies [45,46] have the same conclusion. Kaplan and Winder [46] also showed that the effect on miss ratio is very little as the set size increases from four to fully associative.

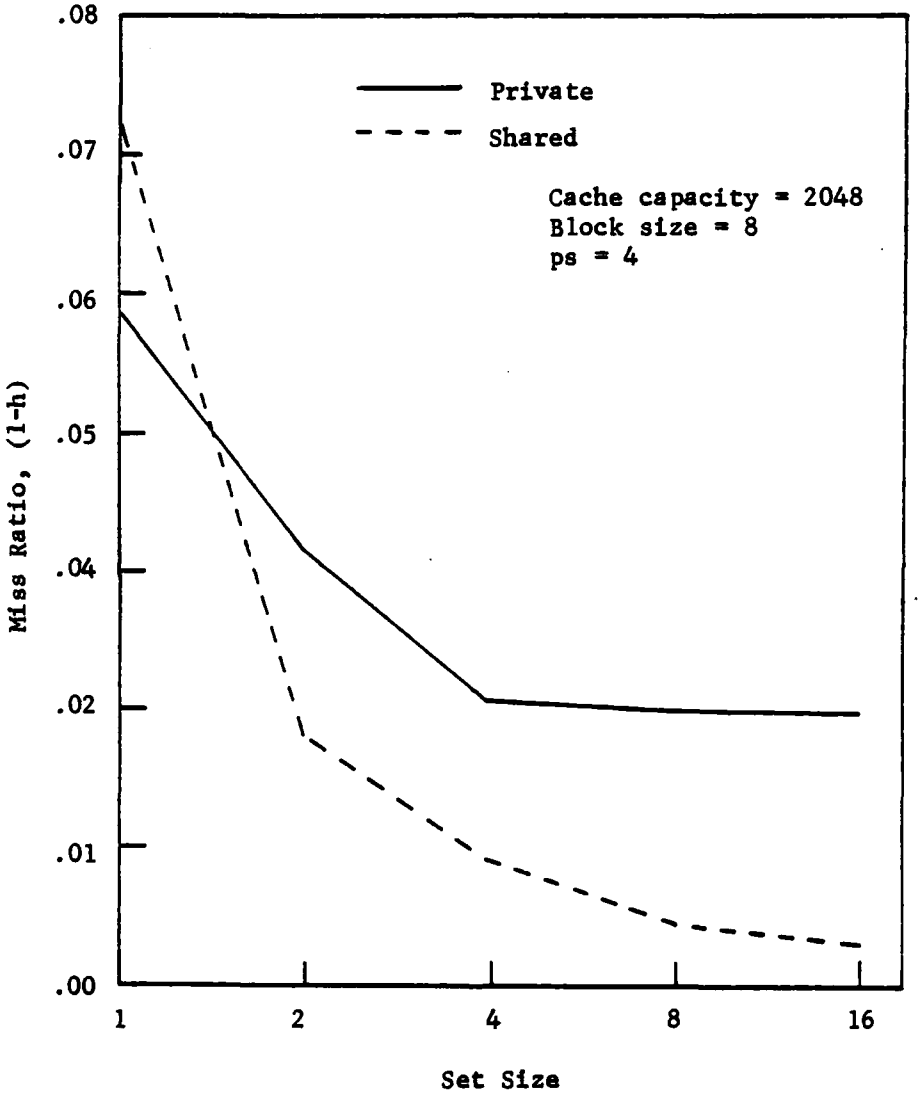


Figure 4.2.3 Effect of set size on miss ratio.

Although not shown here, this asymptotic behavior should be understood because the space contention in each set becomes less for the large set sizes.

As expected, the miss ratio for shared cache is more sensitive to the set size than that for the private cache. This sensitivity is due to the fact that each set in the shared cache is shared by all streams in the system. Since the space contention within each set for shared cache is more severe than that for private cache, shared cache performs worse than private cache for set size equal one, i.e. direct mapping. Note that the deadlock situation caused by space contention within a set, as described in section 2.2, does happen in the experiment if a conventional LRU, instead of the modified LRU, replacement algorithm is used in the shared cache when set size equals one. However, figure 4.2.3 shows that shared cache always performs better than private cache as long as set size is larger than one. The miss ratio of shared cache tends to reach the asymptotic, i.e. fully associative, value at larger set sizes. In figure 4.2.3, the relative miss ratios of shared cache with respect to private cache vary from 1.233 to 0.443 as the set size increases from 1 to 16. Therefore, from the performance point of view, relatively larger set sizes are preferred in the shared cache systems.

4.3 The Effect of Operating Environment and Write Policy on Miss Ratio

In general, cache performance is highly affected by program characteristics. However, the effects of program characteristics on miss

ratio for shared cache are unclear. It was assumed that the write through updating scheme is used for shared cache, but the effect of write through on performance is also unclear. In this section, the effects of different operating environments, write policies, and program characteristics on miss ratio are investigated. The following studies are based on the observations of our simulation experiments. More research will be suggested for those phenomena for which no firm conclusion is obtained from observations.

Assume that there are four streams in the system, i.e. sp=4. An IIID operating environment with all streams being executed having similar program characteristics can be simulated by assigning each processor a different section of a particular program trace and an offset constant to create disjoint address spaces. For convenience, let x be an integer number such that $1 \leq x \leq 4$. Then the four program traces can be specified by the distinct values of x as follows: x=1 for CCOBOL, x=2 for GAUSS, x=3 for ECOBOL, and x=4 for EIGEN. Let y represent different write policies and operating environments. For shared cache, the operating environments of IIID with write back, IIID with write through, and SIID with write through are specified by setting y equal to A, B, and C, respectively. Then, various experiments can be specified by (x,y). Figures 4.3.1 and 4.3.2 illustrate the effects of write policies and operating environments on miss ratio. In these graphs, all four streams are selected from a particular program trace specified by x for each experiment. The dotted lines show the performance for private cache with an IIID operating environment and write back updating scheme.

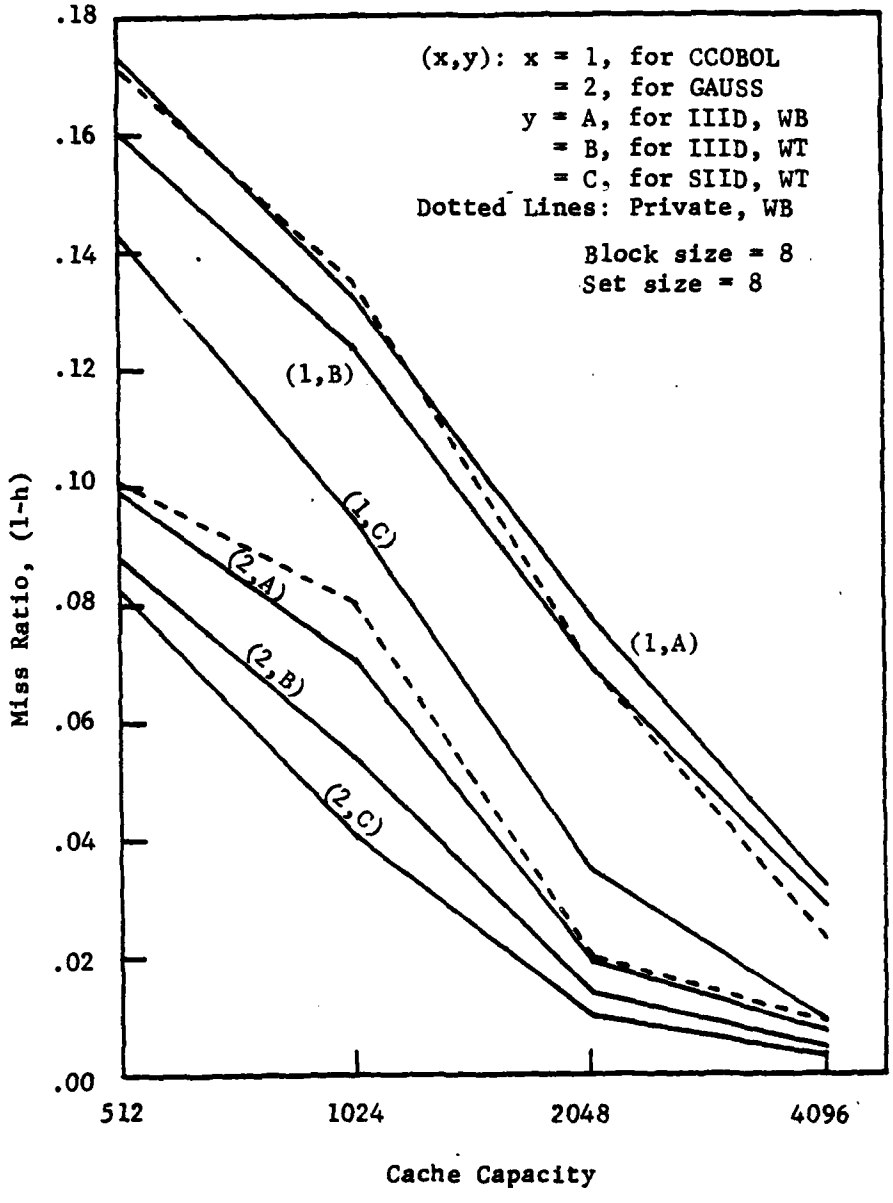


Figure 4.3.1 The effects of write policies, space sharing and operating environments on miss ratio for CCOBOL and GAUSS.

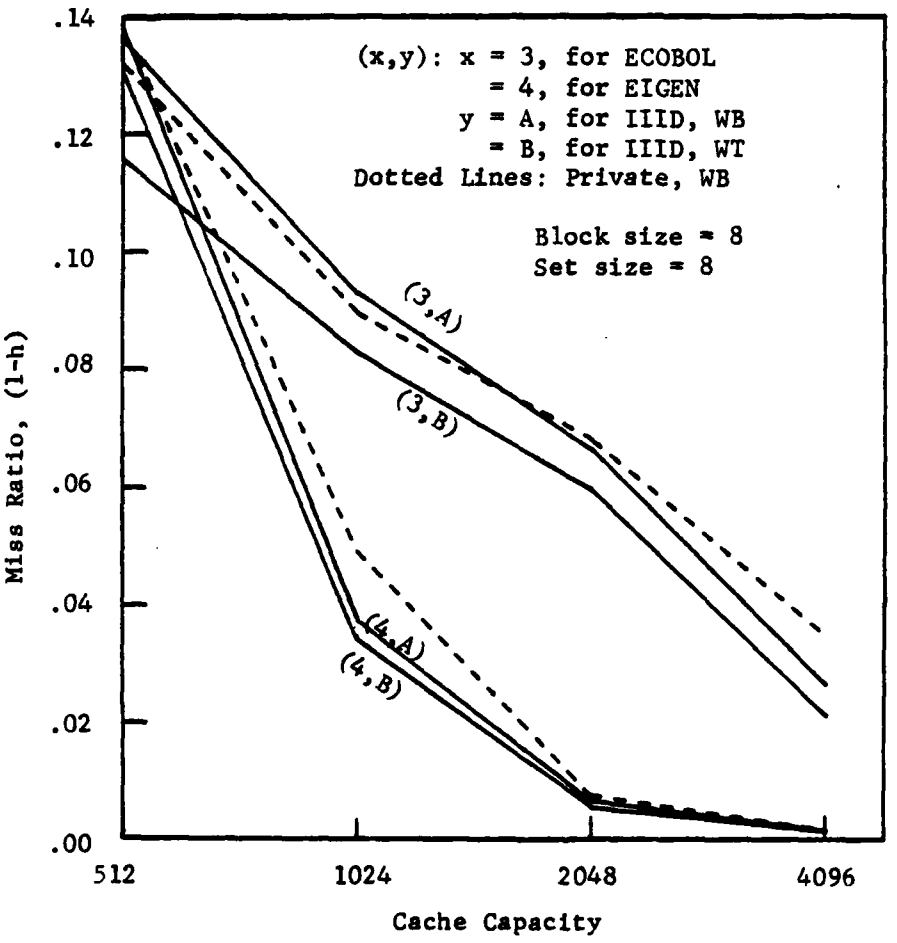


Figure 4.3.2 The effects of write policies and space sharing on miss ratio for EIGEN and ECOBOL.

Note that the difference between (x,A) and (x,B) for a given x is the miss ratio improvement of write through over write back. The percentages of write accesses for EIGEN, GAUSS, ECOBOL, and CCOBOL are 8.8%, 9.2%, 28.6%, and 13.1%, respectively. As can be seen, for shared cache, write through always performs better than write back. However, the amount of miss ratio improvement by write through is not proportional to the amount of write accessing. In the case of a write access followed by some read access within a very short period of time, write through may not improve the miss ratio if both write and read accesses reference the same block. Hence, the performance for write through may depend not only on the amount of write accessing but also on the strategy used for storage allocation.

The effect of dynamic space sharing on miss ratio can be shown by the miss ratio difference between (x,A) and private cache, i.e. the dotted lines. Clearly, shared cache may not always perform better than private cache. The diagrams show that space-sharing is good for GAUSS and EIGEN but not for ECOBOL and CCOBOL. For GAUSS and EIGEN, the largest miss ratio improvement due to space-sharing occurs at a cache capacity of 1K. However, in most cases, the shared cache with write through performs better than the private cache with write back.

Recall that a SIID operating environment is simulated by assigning each processor its own trace section of a particular program trace and an associated offset constant. The offset constant is added only to the data addresses in the associated trace section to generate effective addresses such that the effective data address spaces of the processors

are disjoint. Then the difference between (x,C) and (x,B) for a given x illustrates the effect of shared code on miss ratio. Figure 4.3.1 also shows this effect for GAUSS and CCOBOL. Sharing of programs significantly reduces the miss ratio for these two program traces even with different sections. This significant performance improvement may be caused by many commonly used subroutines or by some common subroutines which are used very often in both programs. Simulation was performed to measure the extent of sharing. After a block is loaded into the cache by a process, it may be referenced by other processes. Such references, namely, references to a block by processes which did not load that block in the first place, were measured. All such references to all blocks were measured as a percentage of total number of hits. These percentages are 25.2%, 27.2%, 30.4%, and 30.4% for GAUSS with 512, 1024, 2048, and 4096 cache capacities respectively. The percentages are 7.5%, 12.2%, 19.8%, and 23% for CCOBOL with 512, 1024, 2014, and 4096 cache capacities respectively. The graph shows that the amount of miss ratio improvement due to shared code is not totally dependent on the amount of shared code. GAUSS has a higher percentage of requests referencing the shared blocks than that of CCOBOL. However, the amount of miss ratio impovement due to shared code is larger for CCOBOL than that for GAUSS. This result may be explained by noting that the miss ratio improvement due to shared code is also dependent on the distribution of the references to the shared code. A larger amount of shared code may not improve the miss ratio significantly if the intervals between references to the same shared block are very large. In addition, the miss ratio improvement due to shared code is dependent on the number of blocks required to duplicate

code for private cache systems.

Figure 4.3.3 shows the miss ratio comparisons between shared cache with different write policies and private cache with write back for the workload of mixed program traces. In this case, the four streams in the IIID operating environment are formed by selecting the first trace section from each program trace. As can be seen write through performs significantly better than write back for all cache capacities. This graph also shows that space-sharing is better than fixed space allocation. Although it seems that a mixed program workload is more suitable for shared cache than the workload with all streams having similar program characteristics, it is too early to reach a firm conclusion because the interaction between program localities under dynamic space sharing are still unclear.

Table 4.3.1 illustrates the effect on miss ratio of an increase in the number of streams as the cache capacity is also increased proportionally. The last column, trace section, in table 4.3.1 indicates which trace section of each program trace is used. Hence, trace section=1 for sp=4 means the first trace section of each program trace is used, 2 means the second trace section of each program trace is used. For sp=8 and trace section=1, the first trace section of each program trace is used to form four distinct streams. However, the other four streams are also formed by the same four trace sections but with distinct offset constants to create disjoint spaces. Both the first and second trace sections of each program trace are used to form distinct streams for sp=8 and trace section=(1,2). Note that the IIID operating

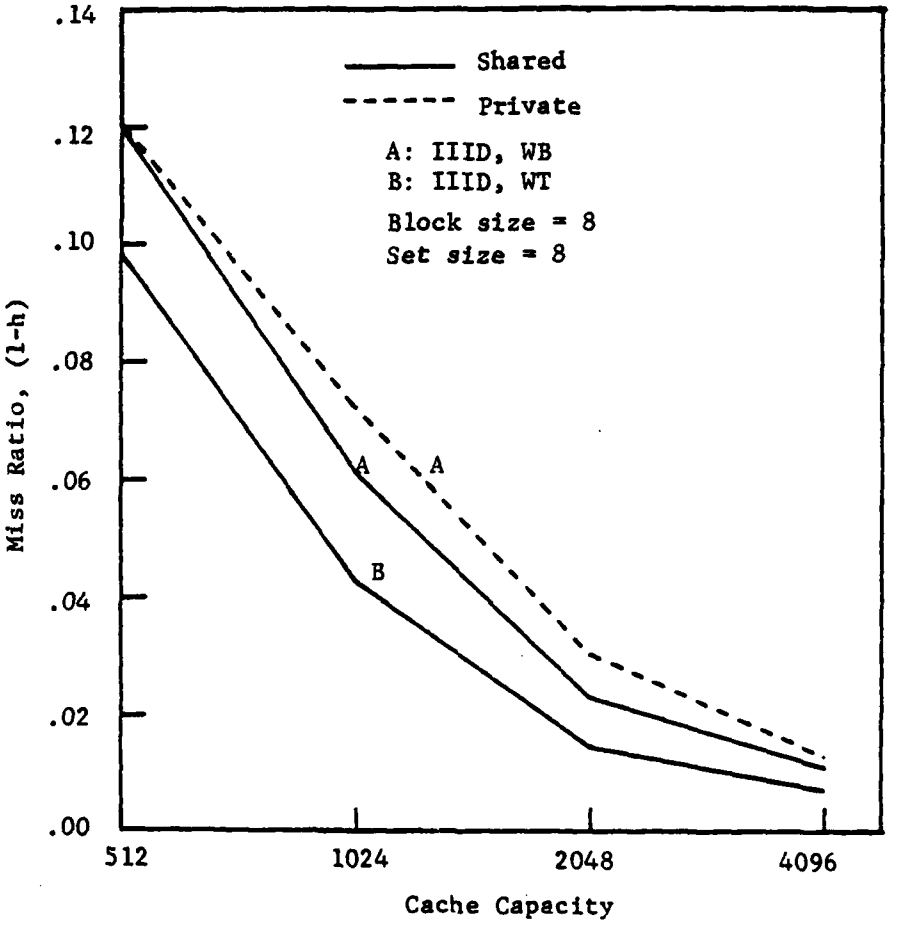


Figure 4.3.3 Miss ratio comparisons between shared cache and private cache for workload of mixed program traces.

Table 4.3.1

The effect of simultaneous increasing both cache capacity and number of streams on miss ratio.

p	s	Set Size	Cache Capacity	Shared/ Private	(1 - h)	Trace Section
1	4	8	1024	shared	.043	1
1	4	16	1024	shared	.041	1
1	4	8	2048	shared	.015	1
1	4	16	2048	shared	.013	1
1	4	8	1024	shared	.089	2
1	4	16	1024	shared	.088	2
1	4	8	2048	shared	.043	2
1	4	16	2048	shared	.042	2
1	4	8	1024	private	.072	1
1	4	16	1024	private	.074	1
1	4	8	2048	private	.030	1
1	4	16	2048	private	.030	1
1	8	8	2048	shared	.046	1
1	8	16	2048	shared	.039	1
1	8	8	4096	shared	.016	1
1	8	16	4096	shared	.013	1
1	8	8	2048	shared	.067	(1.2)
1	8	16	2048	shared	.065	(1.2)
1	8	8	4096	shared	.027	(1.2)
1	8	16	4096	shared	.026	(1.2)
1	8	8	2048	private	.085	( )
1	8	8	4096	private	.038	( )

environment is considered for all experiments in table 4.3.1. Also, write through is used for shared cache and write back is used for private cache.

Although shared cache always performs better than private cache, a simultaneous increase in both cache capacity and number of streams above four for shared cache does not improve miss ratio any further. For example, the average value of miss ratios for four streams obtained from row 1 and row 5 is 0.066, but the corresponding miss ratio for eight streams, row 17, is 0.067. Note that shared cache may perform better than private cache because space-sharing can yield some benefit if some processes need large spaces while other processes need small spaces. However, the space occupied by a process is a function of time. No performance improvement or even worse performance may result if a large space is simultaneously desired by several processes.

The space allocated to a process according to its program locality cannot be easily measured in our simulations. However, an indirect measurement, such as the changes in hit ratio over time, may be used to infer the changes of space desired. From an increase of hit ratio during the interval $(t, t+\Delta t)$ we may infer that fewer blocks of space are required in this interval. Similary, if hit ratio decreases we may infer that more blocks are required. Since a high hit ratio may imply either the process occupies a small number of blocks because it is in a small tight loop or the process occupies a large number of blocks and most required information is already in the cache, the blocks of space allocated to a process cannot be inferred from its hit ratio. However

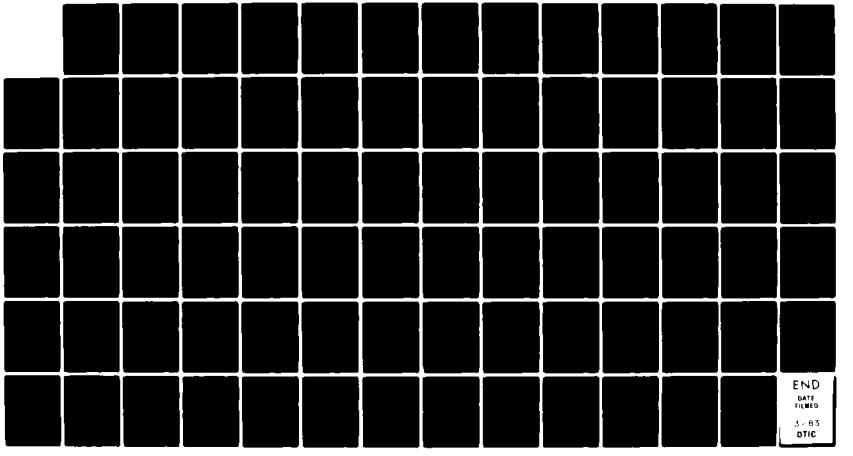
the changes of hit ratio for a process may show the block space needed by this process. If the hit ratio increases for a process while the hit ratio decreases for another in the same time period, space-sharing may result in some benefit during that time period. Since the modified LRU replacement algorithm in each set takes advantage of program locality, the block replaced by some process should be unlikely to be referenced in the near future by its original owner process.

Figure 4.3.4(a) shows the hit ratio vs. time for a private cache of 256 words. Separate experiments have been carried out for the first trace section of both EIGEN and GAUSS. In figure 4.3.4(b), the solid line curve illustrates the hit ratio obtained by using the same two trace sections for a shared cache of 512 words and the dotted line curve shows the average value of the two hit ratios shown in figure 4.3.4(a). Note that time is measured by the number of references. For private cache, the observation period is 100 references and the hit ratio at each observation point is the fraction of requests resulting in hits within previous 400 references. However, for shared cache, the observation period and the average period are double. Since shared cache executes interleaved instructions from two streams, the time moment t in figure 4.3.4(a) corresponds to the time moment 2t in figure 4.3.4(b). In order to highlight the effect of space-sharing on hit ratio, we use write back, fixed block size (=8), and fixed set size (=4) for both shared cache and private cache.

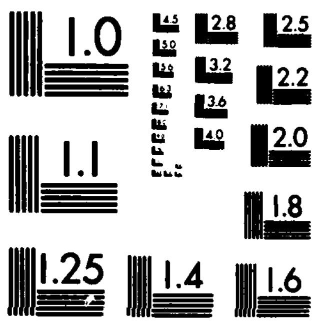
As shown in figure 4.3.4(b), shared cache performs better than the average of two private caches during the periods A, C, and D. During

AD-A124 387 SHARED CACHE ORGANIZATION FOR MULTIPLE-STREAM COMPUTER SYSTEMS(U) ILLINOIS UNIV AT URBANA COORDINATED SCIENCE LAB C YEH JAN 81 R-904 N00039-80-C-0556 3/3

UNCLASSIFIED F/G 9/2 NL



END
DATE FILMED
3-83
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

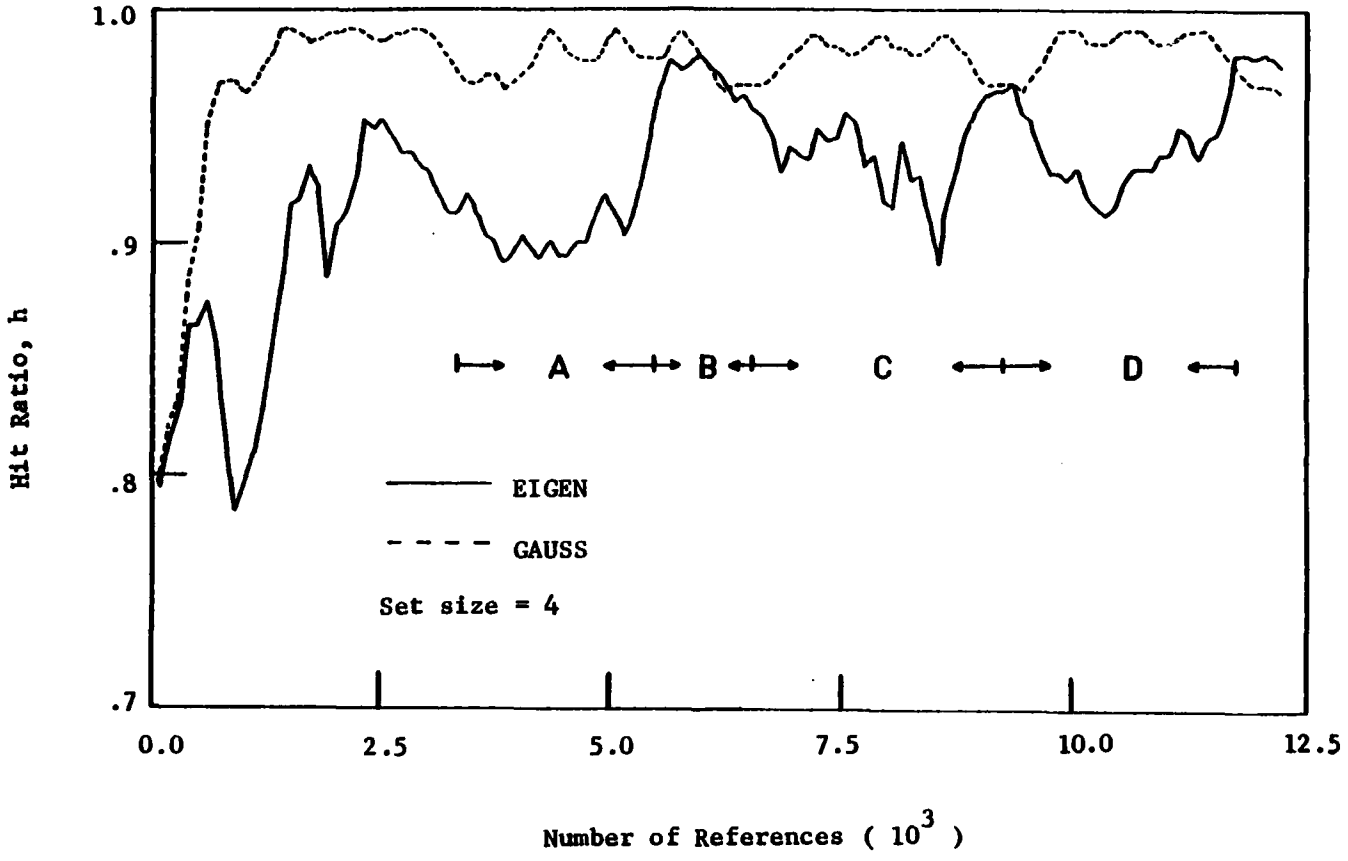


Figure 4.3.4(a) Hit ratio vs. time obtained from private cache for EIGEN and GAUSS.

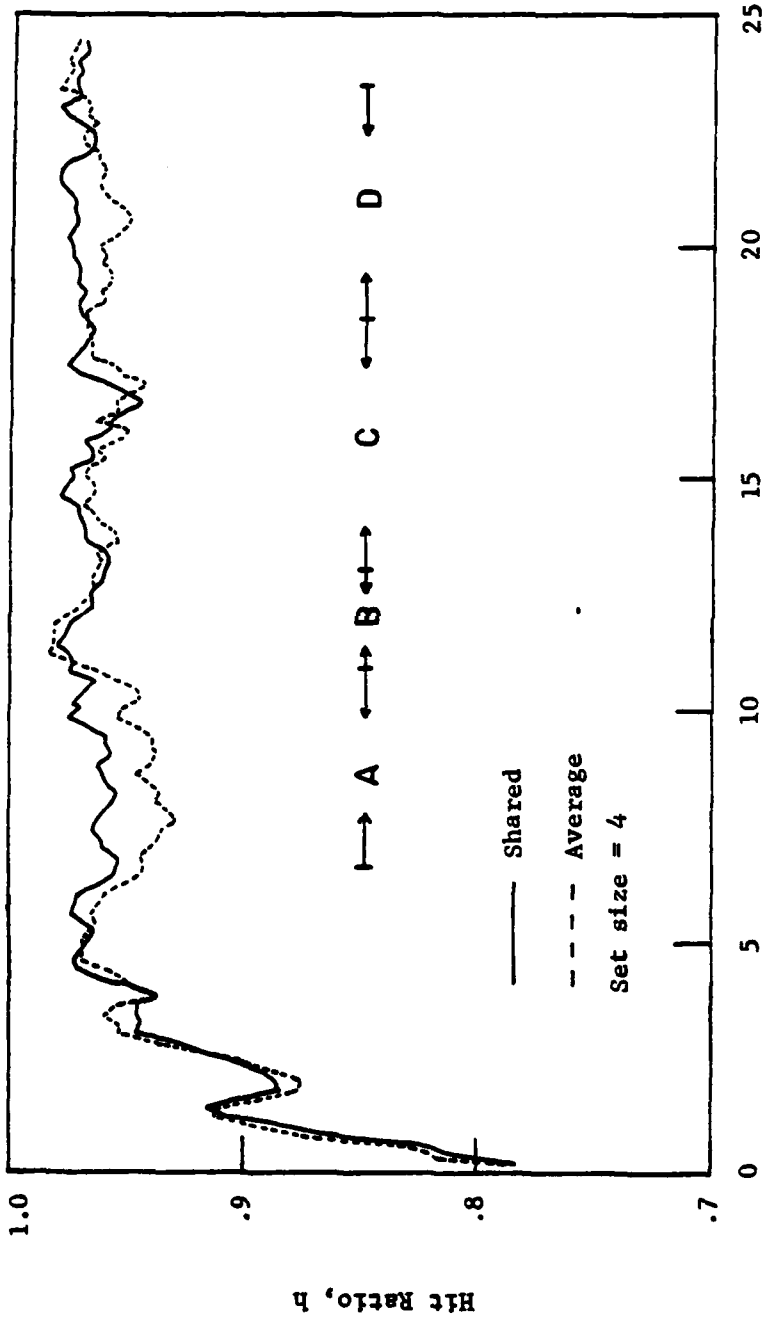


Figure 4.3.4(b) Hit ratio for shared cache and the average hit ratio for figure 4.3.4(a) vs. time.

period B, shared cache performs slightly worse than the average of two private caches. In figure 4.3.4(a), it is seen that during most of periods A, C and D one hit ratio goes through a peak while the other hit ratio goes through a valley, i.e. positive slopes tend to match negative slopes. During most of period B in figure 4.3.4(a), both hit ratios go down. Since both streams require more blocks of space during period B, a worse hit ratio for shared cache may occur if one block replaced by some process is referenced soon by its original owner process. Note that the shared cache gives almost the average hit ratio during the first few thousand references. This coincidence occurs since the cache is empty initially. During the initial period, the high miss ratio is primarily caused by filling the cache, instead of by space contention, and the space-sharing has less effect on performance. In this example, the stationary miss ratio averaged over 25 thousand references for shared cache is 0.044 which gives a 13.1% improvement over private cache (miss ratio=0.051).

Similar experiments have been done for the first and second trace sections of EIGEN. The hit ratios vs. time are shown in figures 4.3.5(a) and 4.3.5(b). As can be seen in figure 4.3.5(b), shared cache performs worse than private cache during the periods A, C, and E in which figure 4.3.5(a) shows that both hit ratios go through peaks. Figure 4.3.5(b) also shows that shared cache is slightly better than private cache in periods B and D in which figure 4.3.5(a) shows that one hit ratio goes through a peak while the other hit ratio goes through a valley. During period F, one hit ratio diagram goes down while the other

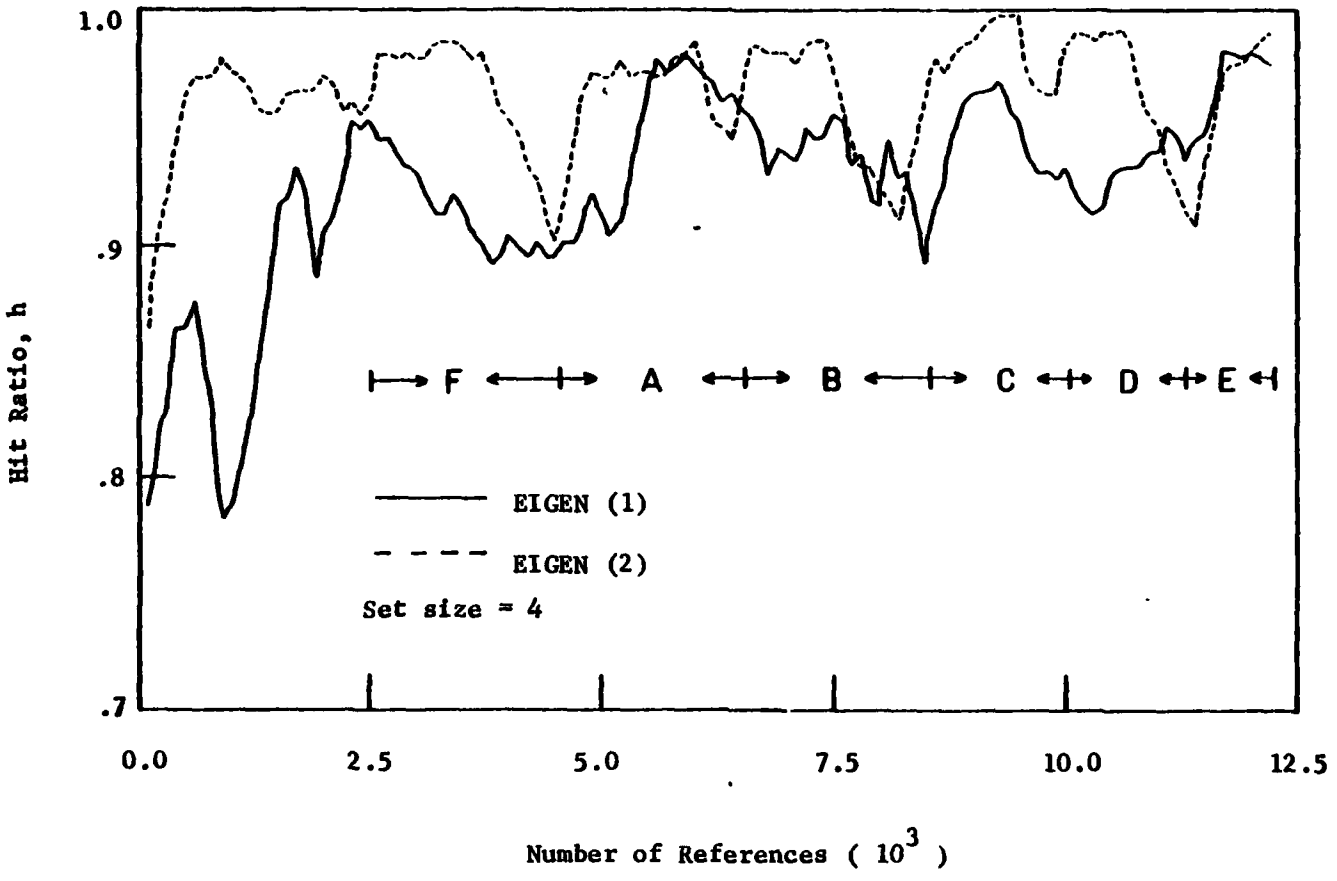


Figure 4.3.5(a) Hit ratio vs. time obtained from private cache for both the first and second trace sections of EIGEN program trace.

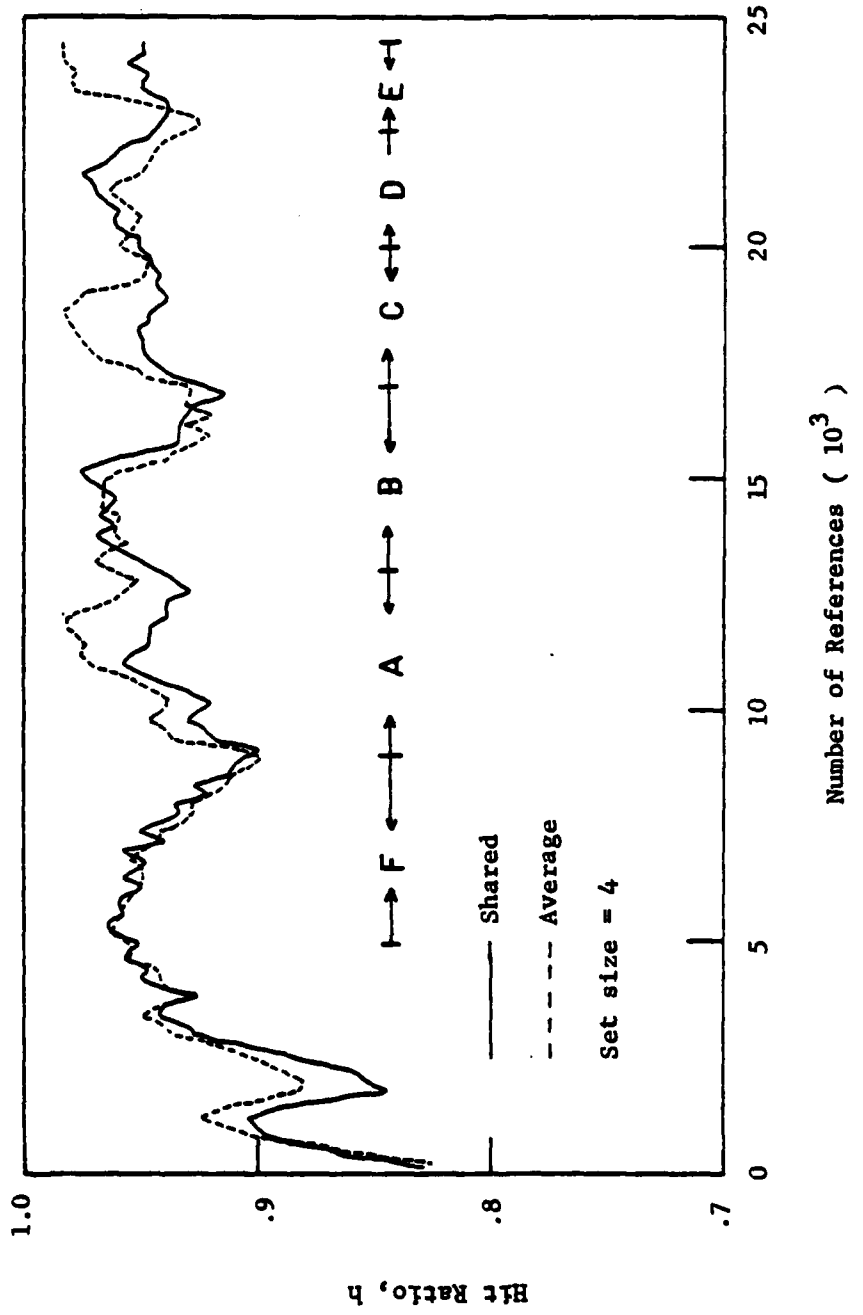


Figure 4.3.5(b) Hit ratio for shared cache and the average hit ratio for figure 4.3.5(a) vs. time.

keeps almost constant as seen in figure 4.3.5(a). An almost average value is seen in figure 4.3.5(b) for shared cache during the same period. The stationary miss ratio averaged over 25 thousand references for shared cache is 0.063 which is 13.2% worse than the private cache (miss ratio=0.056).

Figure 4.3.6(a) and 4.3.6(b) show experiments using the first trace section of both EIGEN and CCOBOL. The shared cache results in almost the same hit ratio as private cache. This coincidence occurs since the CCOBOL program results in large variations in hit ratio during relatively short periods of time as shown in figure 4.3.6(a). Many large variations in hit ratio within short time periods may be considered as rapid and frequent changes in program working sets. In this case, the blocks of space allocated to the process change their content rapidly and frequently. Then the effect of space-sharing on hit ratio is less significant since the hit ratio may be dominated by the changes in program working sets. The stationary miss ratio averaged over 25 thousand references for shared cache is 0.113 which is 4.6% worse than the private cache (miss ratio=0.108). Although not shown here, the hit ratio curve for ECOBOL has a similar shape to that for CCOBOL. This program behavior of rapid and frequent changes in working sets for both CCOBOL and ECOBOL may also explain the reason that shared cache sometimes performs worse than private cache for an IIID operating environment with all streams having CCOBOL or ECOBOL program characteristics as shown in figures 4.3.1 and 4.3.2.

In summary, our experiments show that shared cache may perform

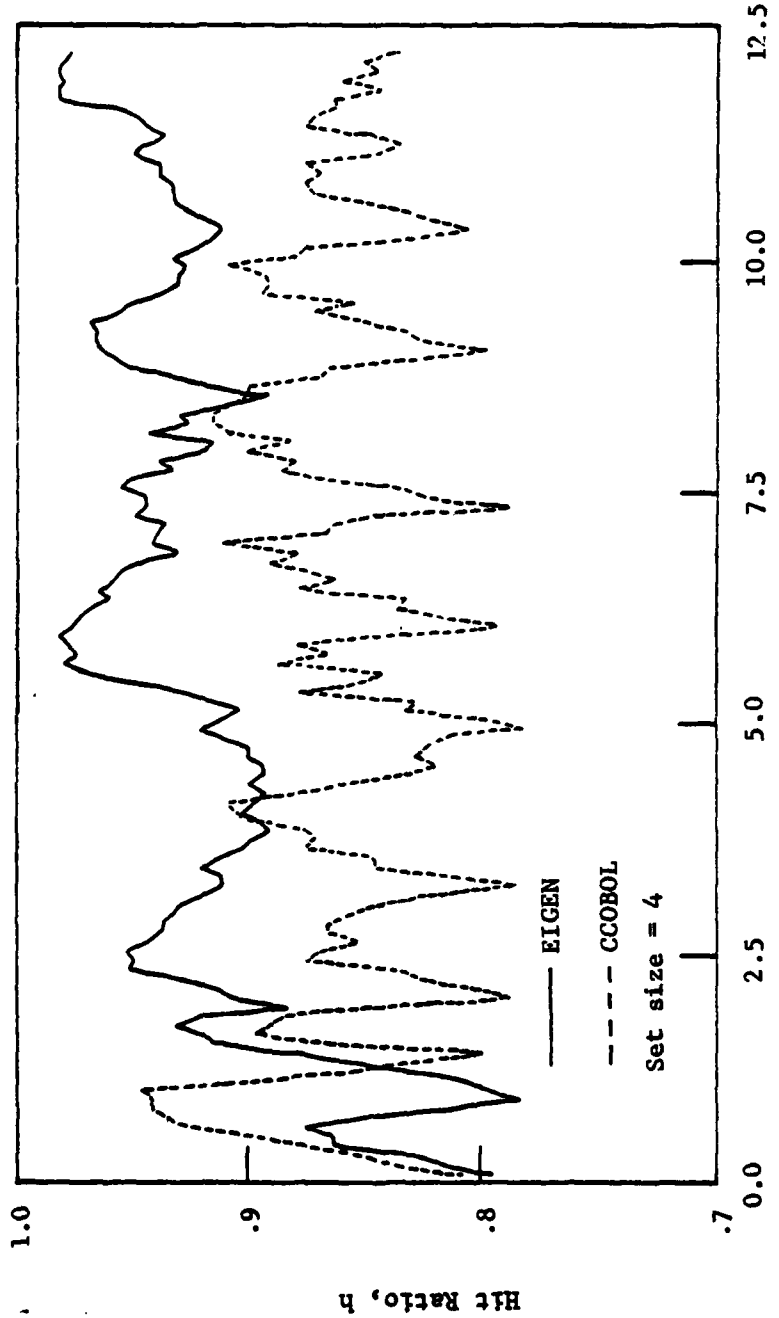


Figure 4.3.6(a) Hit ratio vs. time obtained from private cache for EIGEN and CCOBOL.

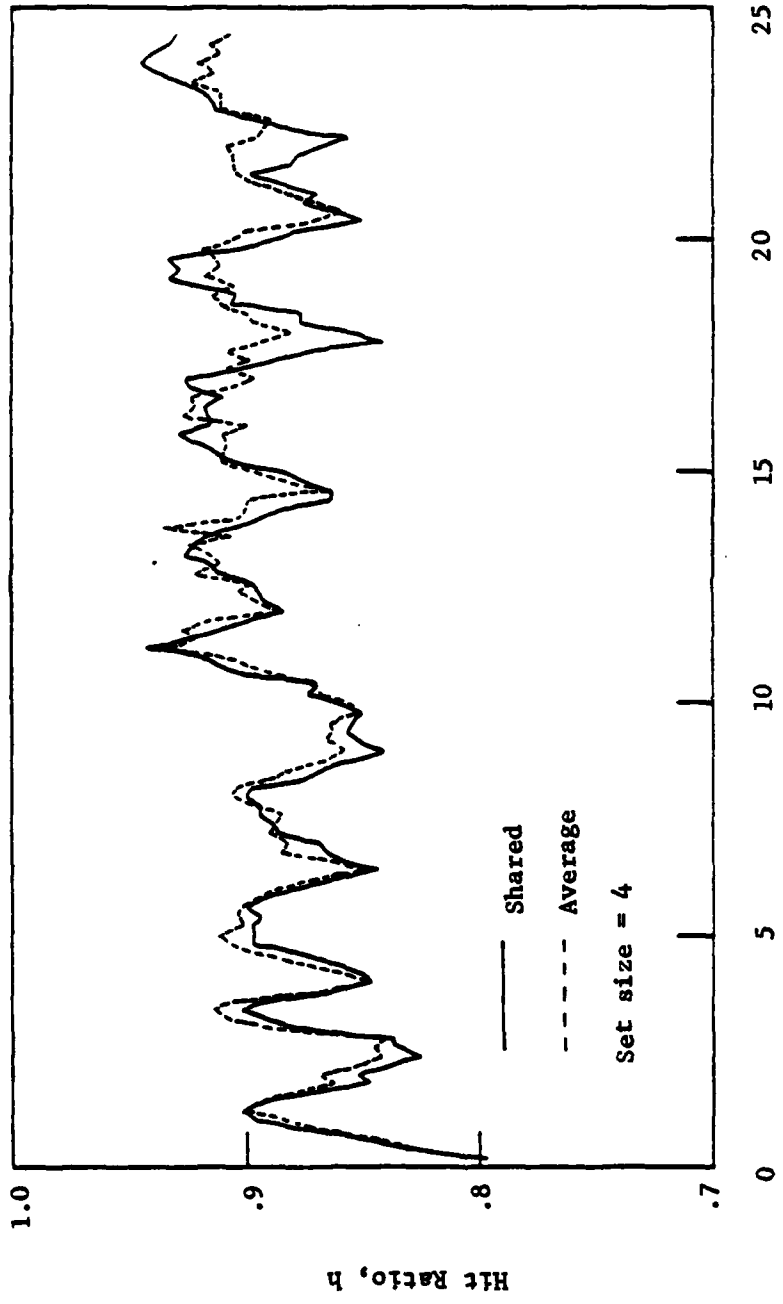


Figure 4.3.6(b) Hit ratio for shared cache and the average hit ratio for figure 4.3.6(a) vs. time.

better than private cache if a workload of mixed programs instead of a workload of all programs having similar characteristics is used. However, space-sharing may not yield any benefit if the mixed program workload contains some streams with the program behavior of rapid and frequent changes in working sets. Experiments also show that shared cache may be better than private cache if a small space need matches a large space need most of the time. No miss ratio improvement or even worse miss ratio may result for shared cache if large space needs happen frequently at the same time for several streams. For shared cache, write through policy always performs better than write back policy. In most cases, shared cache with write through results in a smaller miss ratio than that of private cache with write back.

The above observations are based on the results of our experiments. It was shown that shared cache might perform worse than private cache in some cases since the LRU replacement algorithm in each set could not make an advantage out of space contention. In order to derive an effective shared-cache management policy, more research on dynamic space sharing is necessary.

4.4 Validation of the Models

The effects of various parameters on the miss ratio have been discussed in the previous two sections. In order to isolate the effects of cache access conflict on the miss ratio, the cache cycle, c, and the block transfer time, T, were set to zero and the number of processors, p,

was set to one in previous experiments. In this section, the analytical models for both shared cache and private cache will be verified by comparing the analytical predictions with the simulation results. Due to the simulation costs, it is impractical to verify every possible combination of all parameters for each model. Only several cases for each model will be verified.

For a given hit ratio, the analytical models developed in chapter 3 predict system performance by evaluating cache access conflict. To verify the analytical models, the hit ratio should be treated as a given parameter and only the effects of cache access conflict on performance need to be verified. Hence, the cases chosen to verify the models in this section are the extreme cases of high and low cache access conflict. In figure 4.2.2, a shared cache memory of 1 K cache capacity and block size = set size = 8 results in a miss ratio of 0.043 for four streams. This miss ratio is measured under no cache access conflict. Simulation experiments are repeated for this cache organization to simulate the system of a single pipelined processor with four segments, i.e. p=1 and s=4, under various access conflict situations. A shared cache memory of 2 K cache capacity and block size = set size = 8 results in a miss ratio of 0.067 for eight streams under no cache access conflict, shown in table 4.3.1, is chosen to study the effect of multiple access line collisions on the analytical predictions. This particular cache organization is used to simulate the system of a parallel-pipelined processor of order (2,4). Since the simulation termination time is proportional to the total number of streams in the simulated system, a small number of segments, i.e., 2, is used in some cases to reduce the simulation costs.

In our analytic models, it was assumed that cache hit ratio is independent of cache access conflict and cache memory references are independent and uniformly distributed. These assumptions are verified below by comparing the analytic predictions with simulation measurements. Let the miss ratios measured under no cache access conflict be denoted as $(1-h^*)$ and the miss ratios measured under various cache access conflicts be denoted as $(1-h)$. Let $(1-h_d)$ be the dynamic miss ratio derived from $(1-h)$. Let D_1 represent the ratio of $(1-h)$ to $(1-h^*)$ and D_2 represent the relative performance, CPU utilization, of analytical prediction with respect to simulation measurement. Note that the difference between $(1-h^*)$ and $(1-h)$ shows the deviation of program static miss ratio caused by cache access conflicts. However, $(1-h_d)$ is the dynamic miss ratio for the given program static miss ratio $(1-h)$. The CPU utilization predicted by an analytical model is obtained by plugging $(1-h)$ into the analytic equations for model A. However, for model B, the dynamic miss ratio, $(1-h_d)$, is used to evaluate P_{Ah} and P_{Am} and then performance is obtained by using $(1-h)$, P_{Ah} and P_{Am} in the equation for Cu given by theorem 3.3.2.1.

Table 4.4.1 and table 4.4.2 illustrate both the analytical predictions and the simulation measurements for model A (implicit lookup table) and model B (explicit lookup table), respectively. These tables show that the percentages of deviation of miss ratios due to the cache access conflicts vary from 0.0% to 11.9%. Larger deviations occur at longer block transfer times. In general, $(1-h)$ is less than its corresponding $(1-h^*)$ since reference patterns may be altered due to cache

Table 4.4.1

The effect of cache access conflict
on performance for model A
(set size = block size = 8)

p	s	c	T	ℓ	N	Cache Capacity	$(1-h^*)$	(1-h)	D_1	Cu (Sim)	Cu (Anal.)	D_2
1	4	1	8	1	1	1024	.043	.043	1.000	.722	.699	.968
1	4	1	20	1	1	1024	.043	.043	1.000	.527	.482	.915
1	4	1	8	16	16	1024	.043	.042	.977	.869	.905	1.041
1	4	1	20	16	16	1024	.043	.039	.907	.764	.804	1.052
1	4	2	8	4	8	1024	.043	.042	.977	.718	.770	1.072
1	4	2	20	4	8	1024	.043	.040	.930	.601	.654	1.088
1	4	2	8	4	16	1024	.043	.041	.954	.770	.810	1.052
1	4	2	20	4	16	1024	.043	.040	.930	.632	.680	1.076
4	2	1	8	4	4	2048	.067	.065	.970	.385	.446	1.158
4	2	1	20	4	4	2048	.067	.063	.940	.245	.298	1.216
4	2	1	8	32	32	2048	.067	.063	.940	.672	.734	1.092
4	2	1	20	32	32	2048	.067	.060	.896	.493	.556	1.128

Table 4.4.2

The effect of cache access conflict
on performance for model B
(set size = block size = 8)

p	s	c	T	ℓ	N	Cache Capacity	$(1-h^*)$	(1-h)	$(1-h_d)$	D_1	Cu (Sim.)	Cu (Anal.)	D_2
1	4	1	8	1	1	1024	.043	.043	.043	1.000	.744	.721	.969
1	4	1	20	1	1	1024	.043	.043	.043	1.000	.538	.492	.915
1	4	1	8	16	16	1024	.043	.042	.042	.977	.905	.907	1.002
1	4	1	20	16	16	1024	.043	.040	.040	.930	.785	.802	1.022
1	4	2	8	4	8	1024	.043	.042	.047	.977	.751	.780	1.039
1	4	2	20	4	8	1024	.043	.041	.046	.954	.620	.656	1.058
1	4	2	8	4	16	1024	.043	.041	.049	.954	.798	.818	1.025
1	4	2	20	4	16	1024	.043	.041	.049	.954	.652	.682	1.046
4	2	1	8	4	4	2048	.067	.065	.065	.970	.404	.459	1.136
4	2	1	20	4	4	2048	.067	.064	.064	.955	.254	.301	1.185
4	2	1	8	32	32	2048	.067	.064	.064	.955	.705	.735	1.043
4	2	1	20	32	32	2048	.067	.059	.059	.881	.520	.562	1.081

access conflicts. A miss request under no cache access conflict may become a hit request under cache access conflicts. For example, assume that a reference sequence contains some miss requests followed by a particular request k under no cache access conflict. Assume also that they all reference the same set. Furthermore, assume that request k results in a miss under no cache access conflict because the block referenced by request k is replaced by its former miss requests. Due to the changes of reference patterns under cache access conflicts, request k may be accepted prior to those miss requests by the referenced set (or module) and result in a hit. Hence miss ratio may be reduced under cache acess conflicts. Although the relative deviation of miss ratios can be as high as 11.9%, the relative amount of deviation is small. The high values of the percentage indication are due to the small value of $(1-h^*)$. As can be seen, the percentage of deviation of the hit ratios, rather than miss ratios, for all cases in both tables is less than 1%. The cache access conflicts do affect the cache hit ratios. However, the amount of deviation of the hit ratios due to the cache access conflicts is small. Therefore, the assumption that hit ratio is independent of access conflict should not introduce a significant deviation in the analytic predictions.

The last column in both tables shows the performance of analytical predictions with respect to simulation measurements. The magnitude of D_2 varies from 1.002 to 1.216. In general, the simulation shows lower performance than the corresponding analytic results for $\ell > 1$. In addition, as Cu decreases, the performance difference between the

simulation and analytical results becomes more apparent. In the simulation experiments, the blocked requests are resubmitted with the same addresses one instruction cycle later. Hence the address distribution is not uniform and there is a tendency to reference lines and modules that cause rejections more frequently without success. Therefore, the probability of rejection is higher for the simulation experiments. However, the percentage of performance difference is less than 5.2% if the CPU utilization is higher than 0.77. Therefore, the assumptions that rejected requests are discarded and references are uniformly distributed do not cause a significant deviation of the analytic model from reality for systems with a reasonable performance. Note also that the percentage of performance improvement of model B over model A is less than 5.5% in all the cases listed in both tables.

A simulator for p processors and N main memory modules system with p private cache memories was written by Patel [63]. A random number generator is used to generate both cache and main memory requests. A main memory request is generated immediately after a cache miss has been detected. A blocked main memory request due to access conflict is queued in the buffer, instead of discarded. A buffer is associated with each main memory module. Each instruction cycle an outstanding main memory request is chosen from each nonempty buffer. Assume that there is a cache controller associated with each private cache. After a main memory request is made by a cache controller, this cache controller will not make any new request until the previous request has been served. Table 4.4.3 shows the analytic and experimental results. Note that T^{m}

Table 4.4.3

Performance for private cache systems (ℓ = N)

P	T''	1-h	Cu(Sim)	Cu(Anal.)	D
2	2	.500	.472	.454	1.040
2	2	.250	.647	.640	1.011
2	2	.125	.790	.789	1.001
2	2	.063	.885	.884	1.001
2	2	.031	.938	.941	.997
2	2	.016	.969	.969	1.000
2	8	.500	.162	.172	.942
2	8	.250	.294	.301	.977
2	8	.125	.467	.477	.979
2	8	.063	.658	.653	1.008
2	8	.031	.804	.797	1.009
2	8	.016	.891	.885	1.007
8	2	.500	.436	.446	.978
8	2	.250	.628	.635	.989
8	2	.125	.785	.787	.998
8	2	.063	.885	.884	1.001
8	2	.031	.939	.940	.999
8	2	.016	.969	.969	1.000
8	8	.500	.142	.170	.835
8	8	.250	.260	.301	.864
8	8	.125	.429	.476	.901
8	8	.063	.623	.653	.954
8	8	.031	.785	.797	.985
8	8	.016	.884	.885	.999

represents the block transfer time relative to the instruction cycle and D represents the relative performance of analytic prediction with respect to simulation results. The large performance differences between analytic predictions and simulation measurements occur at very low values of performance. In table 4.4.3, this performance difference is less than 1% for systems with CPU utilization higher than 0.8. Therefore, the deviation of analytical predictions from the simulation results is negligible for reasonably high performance systems.

The effects of cache access conflicts on system performance for a given hit ratio will be studied with various parameters in the following five sections. In order to highlight the effects of cache access conflicts on performance, the effect of cache hit ratio on performance should be isolated. This can be done by choosing a high value of hit ratio. In the following discussion, a hit ratio of 0.98 is chosen for studying the performance degradation due to cache access conflicts. A given hit ratio may be obtained by choosing different combinations of cache capacities, block sizes, and set sizes. However, the effects of these parameters on hit ratio have already been discussed in the previous sections. Hence, it is simply assumed that a fixed hit ratio of 0.98 is given without explicitly specifying the cache capacity, the block size and the set size in the following five sections.

In this section, we have shown that the simulation results were not significantly different from the analytic predictions for reasonably high performance systems. Since we are interested in high performance systems, the following discussions are based only on the analytic

predictions. Also, since the analytic results for model A and model B are almost the same for the range of parameters to be studied, only the analytic results for model A are discussed in the remainder of this chapter.

4.5 Effect of the Number of Cache Modules (N) on Performance

Various cache memory configurations, i.e. (ℓ,m), can be obtained for a given cache capacity. By varying the size of the cache memory module for a given cache capacity, the total number of cache memory modules, N, can be varied. In order to keep hit ratio constant, we assume that the cache capacity is constant for a given p and h. For practical reasons, the total number of cache memory modules cannot be arbitrarily large for a given cache capacity because of the availability of only certain memory chip sizes. However, as the effects of the number of cache modules on the performance are studied, the practical limitation on the memory module sizes is not considered here.

Figures 4.5.1 and 4.5.2 illustrate the effect of N ($\geq \ell$) on CPU utilization for ℓ=4 and 16 respectively. In general, an increase in N increases the performance for given ℓ, p, h, T, and c (>1). For c=1, the number of cache modules, N, has no effect on performance because there is no busy module collision. Recall that the lower bound on P of corollary 3.2.3.1.2 is

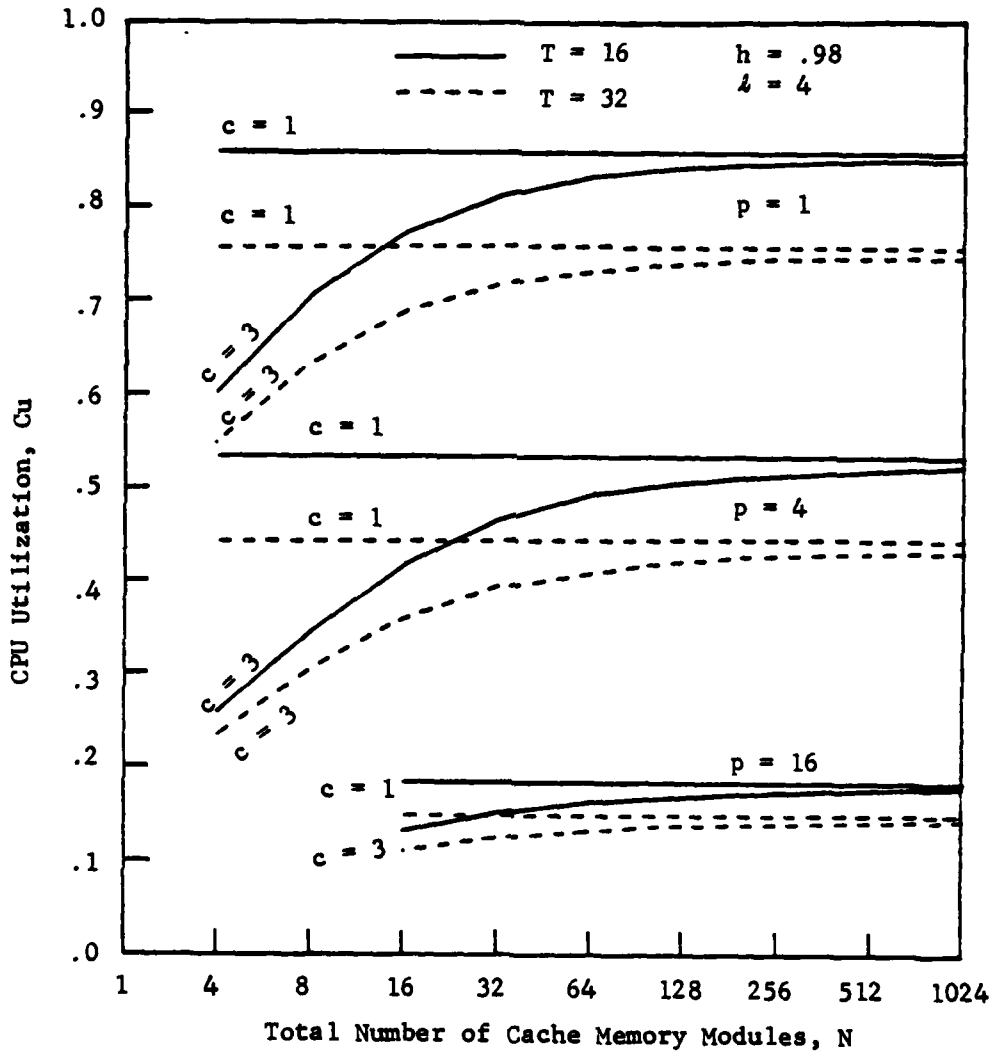


Figure 4.5.1 Effect of N on Cu for $\ell = 4$.

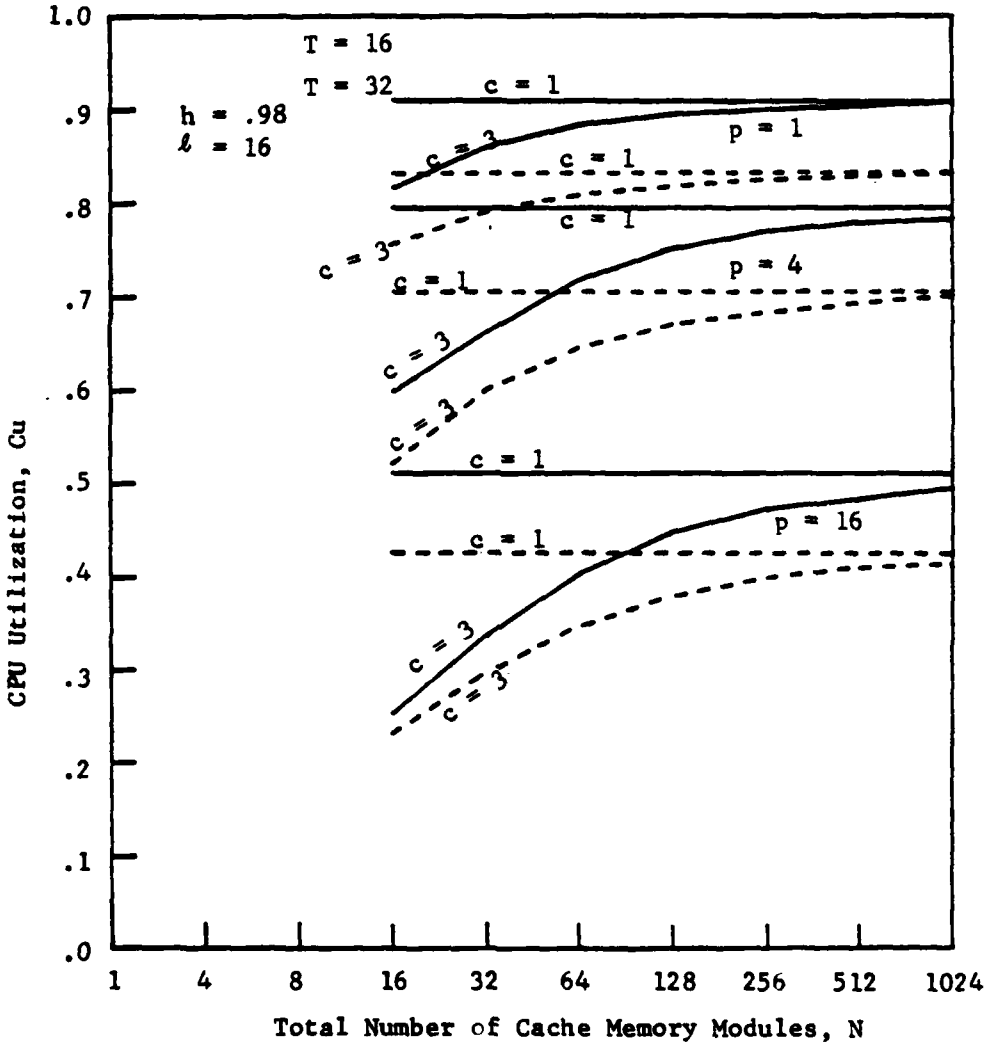


Figure 4.5.2 Effect of N on Cu for ℓ = 16.

$$\frac{\ell N(1-P_1)}{\ell N + Np(1-P_1)(1-h)(T+c-1) + \ell ph(1-P_1)(c-1)}$$

$$= \frac{(1-P_1)}{1 + \frac{p}{\ell}(1-P_1)(1-h)(T+c-1) + \frac{ph}{N}(1-P_1)(c-1)} ,$$

where $1-P_1 = [1-(1-\frac{1}{\ell})^p]\frac{\ell}{p}$.

As N approaches infinity and h is high, the lower bound becomes,

$$\lim_{N \to \infty} P_A = (1-P_1)/[1 + \frac{p}{\ell}(1-P_1)(1-h)(T+c-1)]$$

From the above limiting expression, it is seen that for large N and h, the cache memory cycle, c, does not have a significant effect on P_A or Cu. Hence, for large N and h, Cu is limited by ℓ , p, and T. The graphs also show that the block transfer time, T, highly affects Cu. This effect becomes larger for c 1 as N increases. As an illustration, consider figure 4.5.1, which is for ℓ =4. Suppose that p=1 and Cu is required to be 0.75. Using (c,T)=(3,32), N is required to be at least 256, whereas, if (c,T)=(3,16), N may be as low as 16. In either case, N is significantly larger than pc.

If the block transfer time, T, is primarily deminated by the main memory cycle, i.e. the only way to reduce T is faster main memory, another tradeoff between c and T for each p can be found in the graphs.

A system with slow main memory and fast cache memory may perform better than a system with fast main memory and slow cache memory. For example, consider figure 4.5.2, for which ℓ =16. Suppose that p=16 and cache memory can at most be divided into 64 modules due to practical restrictions on the module size for a given cache capacity. Then the performance obtained by using (c,T)=(1,32) is higher than that obtained by using (c,T)=(3,16). In addition, the cost of a system using (c,T)=(1,32) may be cheaper than the cost of a system using (c,T)=(3,16). Since the size of the main memory is usually much larger than the size of the cache memory, speeding up the main memory may then be much more expensive than speeding up the cache memory. However, the reverse tradeoff will be true for this example if N=128 is allowed.

Figures 4.5.1 and 4.5.2 show that there is progressively less payoff (increase in Cu) from increasing N for large ℓ and small p, and for small ℓ and large p. On the other hand, there is some significant payoff to increasing N as ℓ is close to p in both models.

4.6 Effect of the Number of Lines (ℓ) on Performance

Intuitively, an increase in ℓ reduces multiple access line collisions and busy line collisions and therefore increases performance. Figures 4.6.1 and 4.6.2 illustrate the effect of the number of lines on performance for N=64 and c=3 and for N=1024 and c=1,respectively. The graphs show that poor and undesirable performance occurs in the region $\ell < p$. For $\ell \ll p$, the performance is extremely low because of the

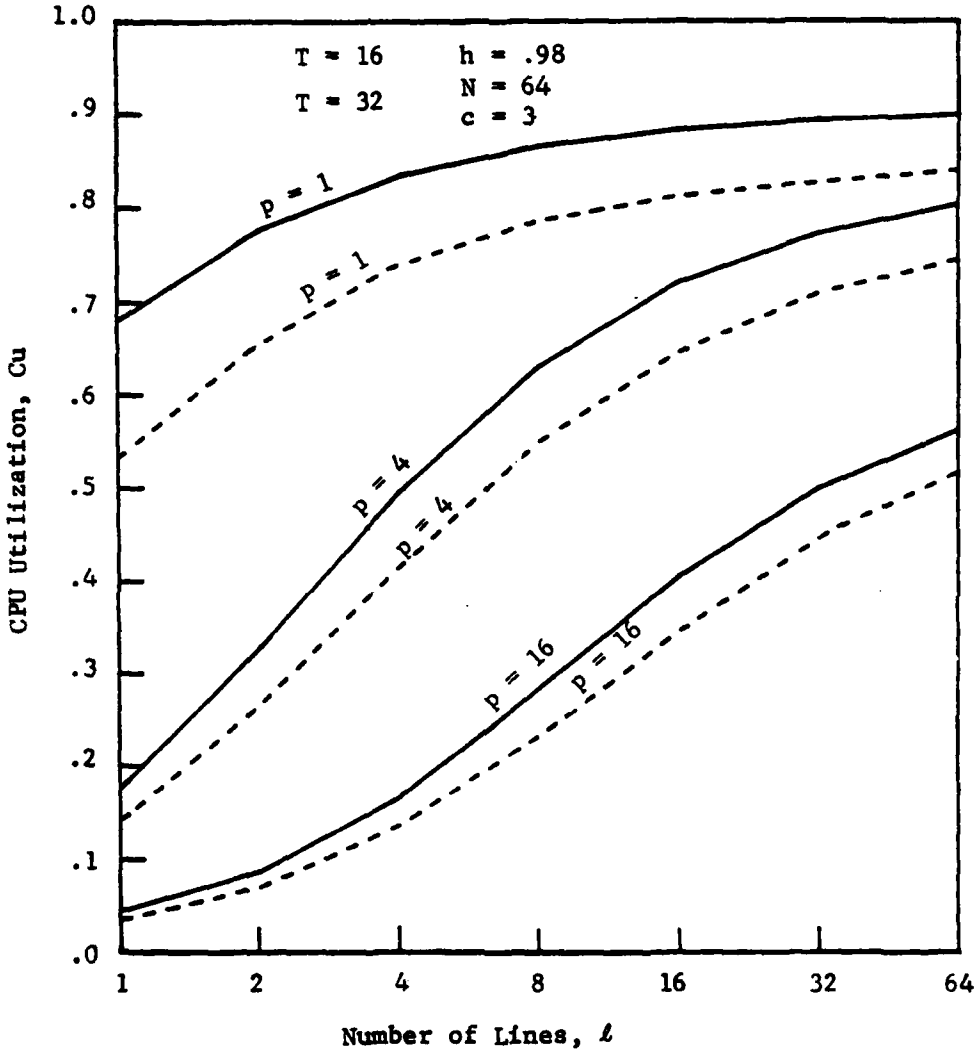


Figure 4.6.1 Effect of ℓ on Cu for N = 64 and c = 3.

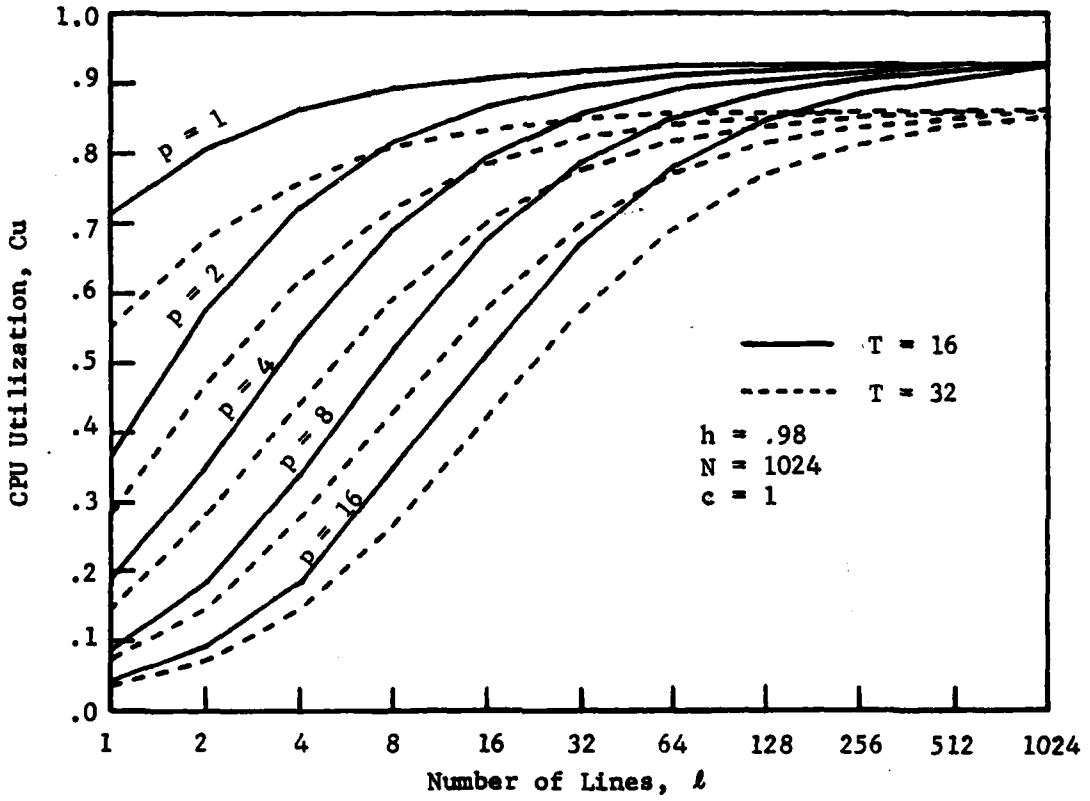


Figure 4.6.2 Effect of ℓ on Cu for N = 1024 and c = 1.

excessive multiple access line collisions. Hence in this region there is very little payoff in performance to doubling ℓ .

There is a point of inflection at ℓ =p. In general, for ℓ in the neighborhood of p, Cu is most sensitive to increases in ℓ and a significant increase in Cu occurs for small increases in ℓ . In the region of ℓ >>p, multiple access line collision probability P_1 is close to 0 and therefore probability of acceptance P_A is close to 1. Hence the performance is limited only by (1-h)T", where $T''=\lceil T/s \rceil$ is the block transfer time in number of processor cycles. There is little payoff in performance to increasing ℓ for small (1-h)T" because the performance is almost saturated. In this case, the asymptote is $Cu = \frac{1}{1 + (1-h)T''}$.

4.7 Effect of Cycle Characteristics on Performance

An increase in the block transfer time, T, increases the busy line collisions and the waiting time for cache misses, whereas, an increase in the cache memory cycle, c, increases busy module collisions. However, these effects are very small when ℓ and N are sufficiently large.

In general, the block transfer time may be a function of the cache cycle time. An increase in c may result in a larger T. However, the block transfer time is also a function of block size, number of modules per line, main memory cycle and main memory bandwidth. Hence, fixed T

can be obtained by adjusting these other parameters when c varies. In this section, the effects of c and T on performance will first be discussed separately by varying each one while holding the other constant. Then, the combined effect of a simultaneous increase in c and T is illustrated. It should be noted that c cannot be arbitrarily large, since the model requires the degree of pipelining, s, to be larger than c.

The effect of T can be explained somewhat analytically by using the lower bound formula, i.e. corollary 3.2.3.1.2,

$$P_A \geq \frac{(1-P_1)}{1 + \frac{p}{\ell}(1-h)(1-P_1)(T+c-1) + \frac{ph(1-P_1)(c-1)}{N}}$$

In order to highlight the effect of T on performance, it is assumed that $\ell \ll N$ and $c \ll T$. Note that T has two effects on performance, namely, miss penalty and busy line collision. For small ℓ such that $\ell \ll p$, $P_A \approx \ell/p$ for high h. The performance degradation in this region is primarily due to the excessive multiple access line collisions. Hence the performance, Cu, is very sensitive to the variations in ℓ but insensitive to the variations in T for $\ell < p$. As ℓ increases to p, $P_A \approx (1-P_1) / [1 + (1-P_1)(1-h)T]$ and therefore the CPU utilization $Cu = 1/[1/P_A + (1-h)T"]$ can be approximated as $Cu \approx 1 / [1/(1-P_1) + (1-h)(1 + 1/s)T]$ for large N. The performance is very sensitive to the variations in both T and ℓ for ℓ near p. In other words, system performance is critically dependent on the effects of miss penalty, busy line collision,

and multiple access line collision for $\ell \approx p$ and large N. For $\ell \gg p$, $P_A \approx 1$ and $Cu \approx 1 / [1 + (1-h)\, T/s\,]$ for high h. The performance is then insensitive to ℓ but sensitive to miss penalty, $(1-h)T^m$. In this region, the effect of access conflict on performance is insignificant and the performance is almost entirely dependent on miss penalty. Hence, the block transfer time, T, affects the performance significantly for $\ell \geq p$. Figure 4.7.1 illustrates the effects discussed above. Note that an increase in c shifts the curves down.

Figure 4.7.2 shows the effects of cache memory cycle, c, on performance. It was shown in section 4.5 that performance is less sensitive to c for large N. The sensitivity for N=64 is illustrated in figure 4.7.2. Figure 4.7.3 illustrates the combined effect of a simultaneous increase in c and T for T/c=8.

In general, if N is large enough, variations in c have little effect on Cu for any configuration. If ℓ is large or ℓ is close to p, variations in T have a significant effect on Cu.

4.8 Effect of the Number of Processors (p) on Performance

The choice of p is very critical to obtaining reasonable performance. The sensitivity of P_A to p can be evaluated for some class of p as below by employing the lower bound formula, i.e., corollary 3.2.3.1.2, for P_A. For $p \gg \ell$, $(1-1/\ )^p \approx 0$ and $(1-P_1) \approx \ell/p$. Hence, the lower bound formula is approximated as $\ell / p[1 + (1-h)(T+c-1)]$, for

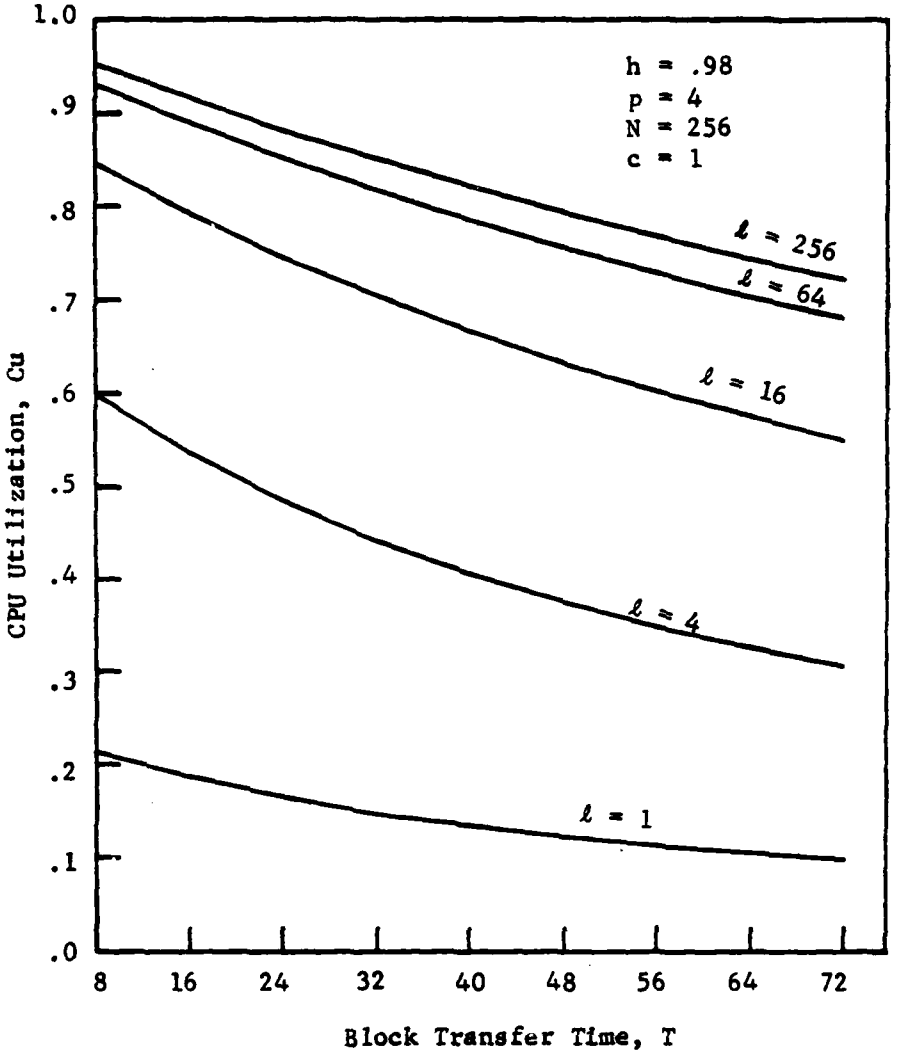


Figure 4.7.1 The effect of T on Cu for N = 256 and c = 1.

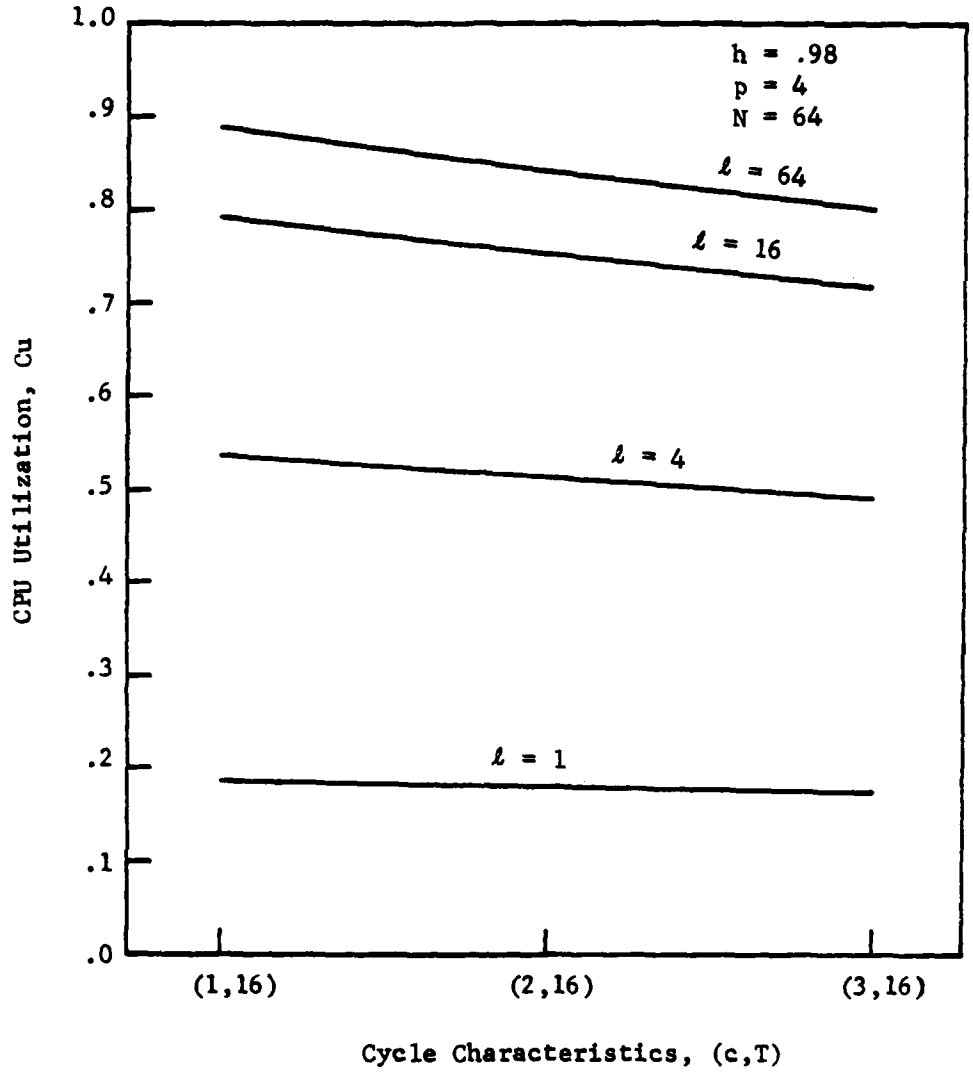


Figure 4.7.2 The effect of c on Cu for N = 64 and T = 16.

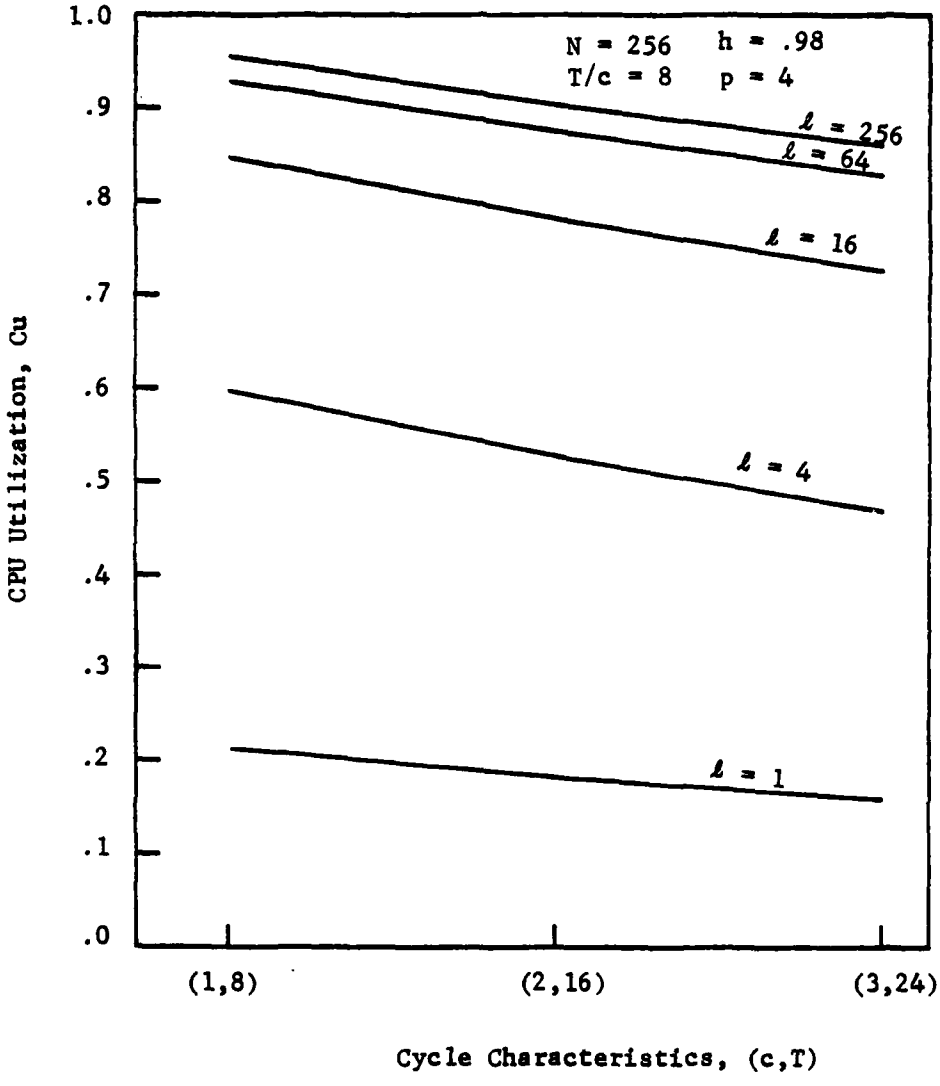


Figure 4.7.3 Effect of (c,T) on Cu for T/c=8 and N=256.

large N. Furthermore, if h is high and T is small, then the lower bound formula is approximately reduced to ℓ/p. Hence, both P_A and Cu are small in this region. Asymptotically, both P_A and Cu decrease to zero if p increases without limit. However, for $p \ll \ell$, $(1-1/\ell)^p \approx 1-p/\ell$ and $(1-P_1) \approx 1$. Therefore, in this region, the lower bound formula for P_A becomes approximately $\ell N / [\ \ell N + pN(1-h)(T+c-1) + p\,\ell h(c-1)]$. For large N and h and small T, P_A is close to 1. An increase in p does not affect P_A and Cu significantly as long as $p \ll \ell$, h is high and T is small.

Figure 4.8.1 illustrates the effects discussed above. Note that Cu is most sensitive to p for p in the neighborhood of ℓ. Some tradeoff between ℓ and T for certain p can be found in figure 4.8.1. For example, given p=8, a system with ℓ =256 and (c,T)=(1,32) results in an almost the same performance as that of a system with ℓ =64 and (c,T)=(1,16). Then the tradeoff between ℓ and T can be determined by the costs of different-sized crossbar switches and different-speed memories.

The total system throughput per instruction cycle, pCu, may be a usuful performance indicator to understand the effects of p on entire system performance. Figure 4.8.2 illustrates this effect for c=1. For $p \ll \ell$, P_A is close to 1 and an increase in p increases the system throughput almost linearly. However, for $p \gg \ell$, the curves become flat and no significant further throughput improvement can be achieved by increasing p.

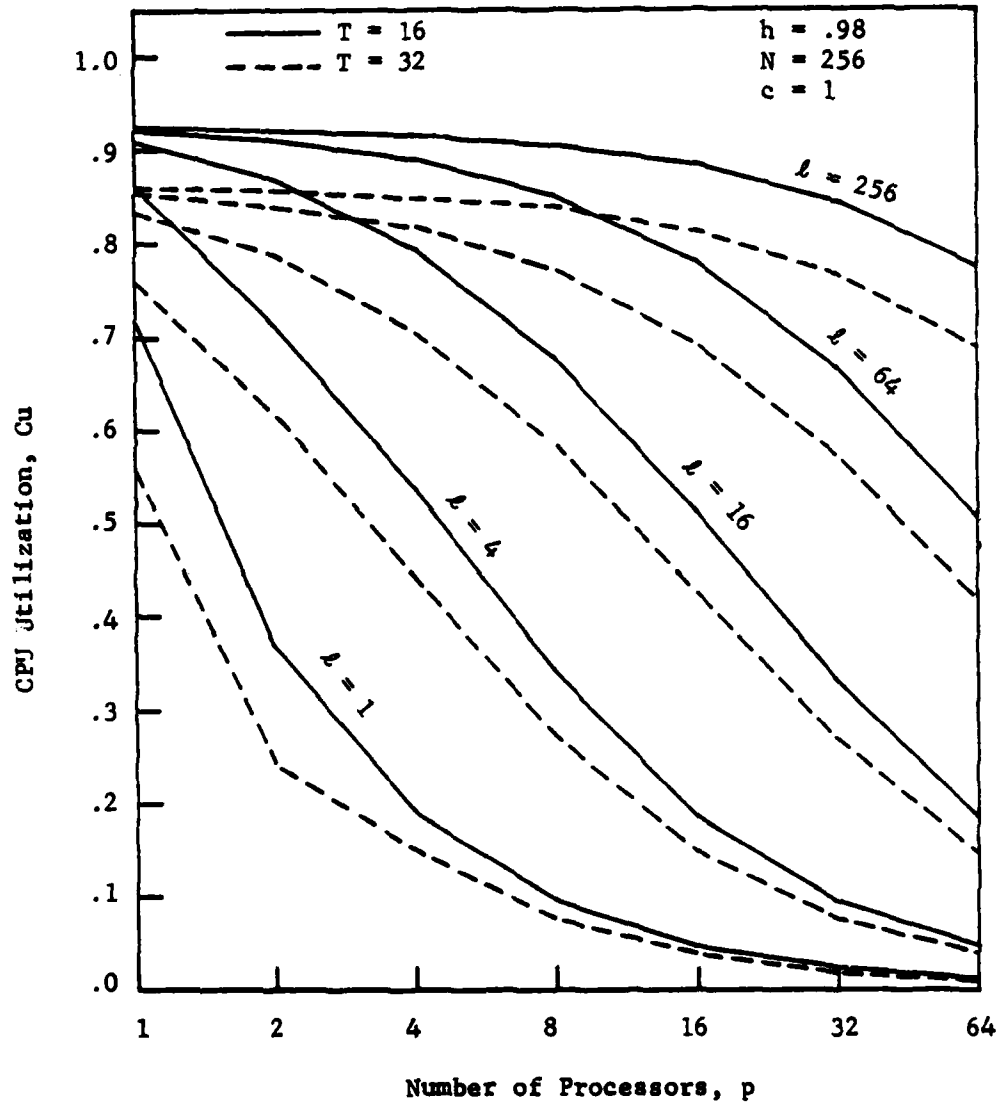


Figure 4.8.1 Effect of p on Cu for N=256 and c=1.

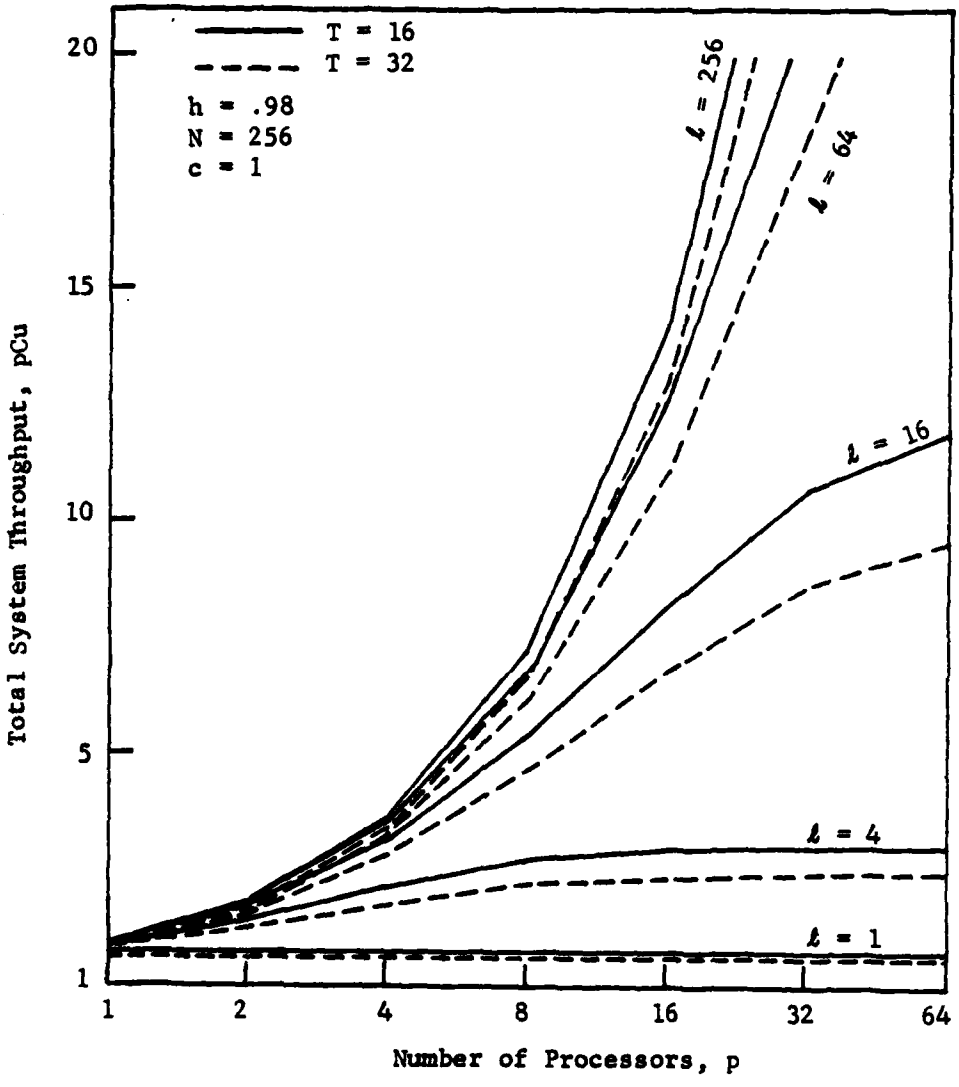


Figure 4.8.2 Effect of p on pc for N=256 and c=1.

Although not shown here, an increase in c shifts the curves down. An increase in N shifts the curves up because decreasing c and increasing N result in a similar effect on performance. Note that the total system throughput is monotonically nondecreasing as p increases.

In summary, P_A is close to 1 and neither P_A nor Cu are very sensitive to small variations in p for $p \ll \ell$. Hence, the total system throughput increases almost linearly with p. As p increases to ℓ, P_A becomes very sensitive to variations in p and there is a point of inflection in both the Cu and pCu curves in this region. Once $p > \ell$, both P_A and Cu decay asymptotically to zero as p increases further. In this region, the total system throughput curves flatten.

4.9 Effect of Processor Speed on Performance

In the analytic models and simulation experiments, the segment time unit, τ, was not explicitly considered. In this section, the effects of processor speed on performance are discussed for three different cases. Recall that the cycle characteristics, (c,T), used throughout this thesis are measured relative to τ. Let (c_0,T_0) denote the absolute cycle characteristics, then $c = \lceil c_0/\tau \rceil$ and $T = \lceil T_0/\tau \rceil$.

Since decreasing τ also increases the rate of requests to the cache memory, Cu is not an appropriate performance indicator of the effect of τ. Instead, the absolute throughput, pCu/τ, is adopted as a measure of performance in this section. Note that the delays due to bussing and

crossbar switching were assumed to be transparent in all models. In this section, it is assumed that these delays are so small that they are still transparent within the range of processor speed to be studied. Since there is no exact solution for P_A in the general case, the lower bound for model A is used in the following discussion.

For the case of a constant request rate assumption, the number of processes which request cache memory within one cache memory cycle is fixed at I_0 as τ varies. Then $I_0 = pc$. Therefore, doubling the speed of processor (halving τ) doubles c and requires that p be halved in order to keep I_0 constant. Figure 4.9.1 illustrates the example I_0=4, h=0.98, (c_0,T_0) = (200,1600)ns, N=256, and p goes from 1 to 4 as τ goes from 50 to 200 ns. Observe that an increase in the speed of the processor (decrease in τ), increases the throughput for configurations with small ℓ. The increase in the throughput for small ℓ is primary due to the effect of fewer multiple access line collisions while reducing p simultaneously as the speed is increased (τ is decreased). Note that lines must be faster (as τ decreases) since address hold time, a, still equals one. For large values of ℓ , an increase in the processor speed, reduces the throughput slightly. Although a reduction in p reduces the multiple access line collisions, the busy line collisions and busy module collisions are increased due to the effect of increasing (c,T) simultaneously as the speed is increased. Therefore, the reduction in throughput for large ℓ as τ decreases is due to a combination of the contrary effects of reducing p while increasing (c,T) simultaneously as the speed is increased. The multiple access line collision is negligible

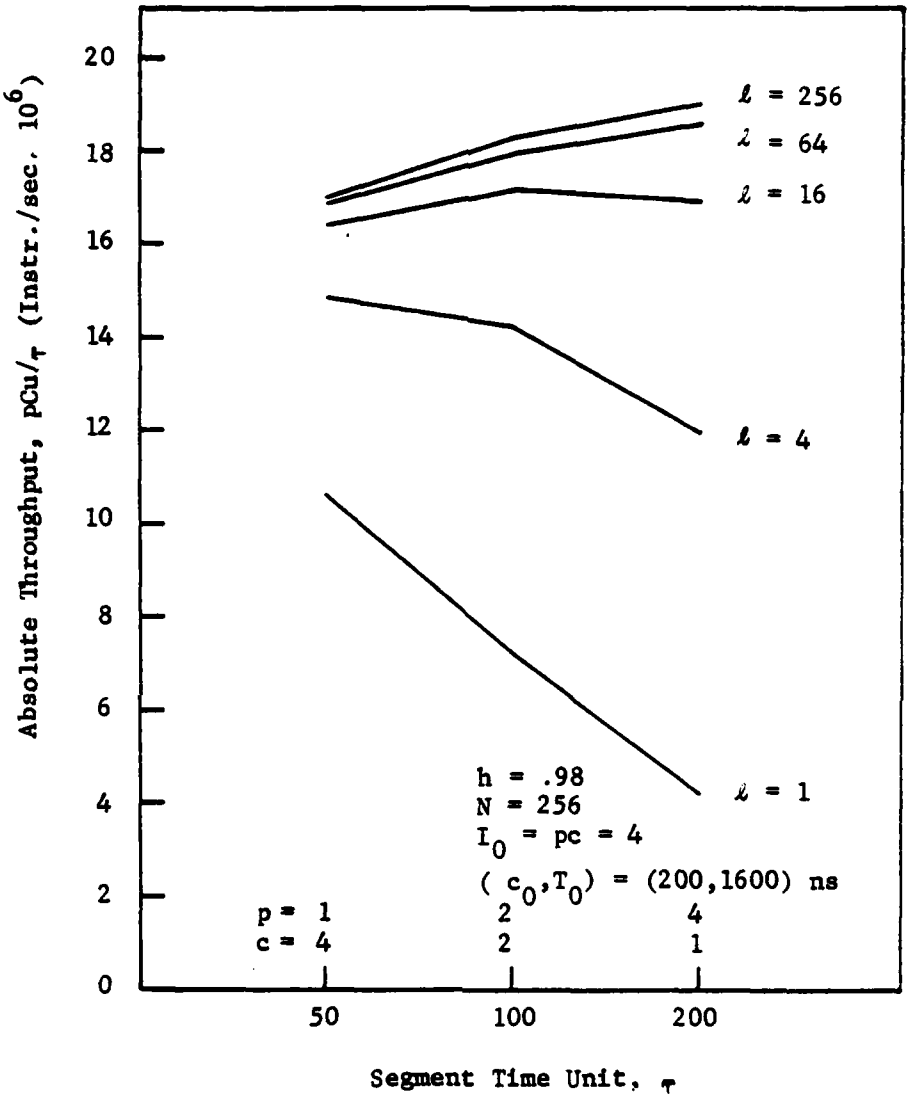


Figure 4.9.1 Effect of processor speed on performance for a constant request rate.

for ℓ >>p, but more busy line collisions and busy module collisions occur for a decrease in p (for constant request rate). Hence the throughput is decreased as p decreases (τ decreases) for ℓ >>p. For the constant I_0 assumption, doubling τ halves s. Hence sp is constant as τ changes. Obviously, a change in τ corresponds to physical changes in the processor design.

Consider the second case in which I_0 is not fixed. Assume that N, p, and (c_0,T_0) are fixed but τ varies. Note that s will still vary inversely with τ and therefore some processor changes are required. In this case, an increase in processor speed increases the degree of pipelining, s, and the rate of requesting cache memory. Figure 4.9.2 illustrates the example for N=256, p=4, h=0.98 and (c_0,T_0)=(200,1600)ns. In general, an increase in the processor speed increases the throughput. Note that an increase in processor speed increases the cache memory bandwidth because the cache request rate is increased. However, this bandwidth increase is faster for larger ℓ because of fewer multiple access line collisions. Hence throughput improvement is relatively higher for larger ℓ as the processor speed increases.

In the third case, it is assumed that I_0, N, s, p, c and T are all fixed. Decreasing τ then simply corresponds to faster clocking of the processor. Furthermore, decreasing τ requires a proportional decrease in (c_0,T_0) so that (c,T) is fixed. For example, let N=256, p=4, h=0.98 and (c,T)=(1,8). Cu is a constant for a given memory configuration as processor and memory speeds change. Figure 4.9.3 shows the effect of processor and memory speed on the throughput for various memory

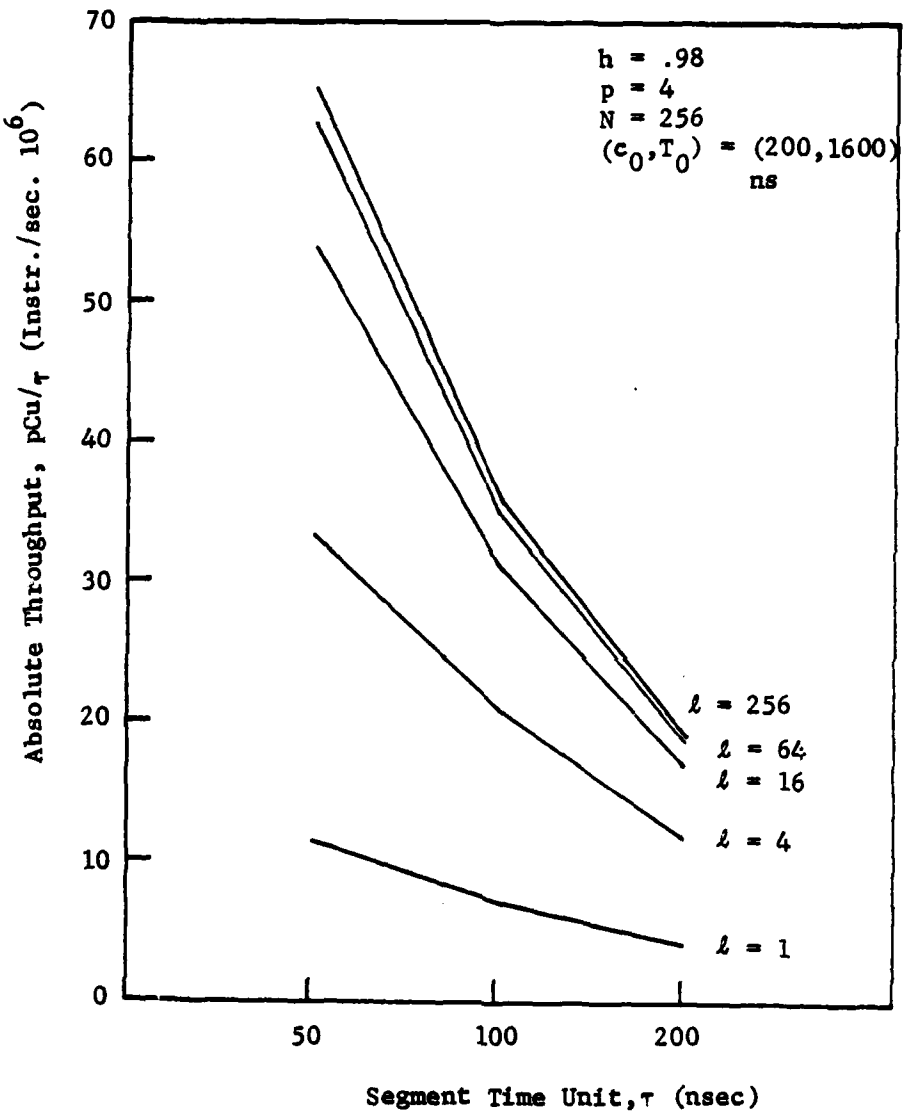


Figure 4.9.2 Effect of processor speed on throughout for varying request rate.

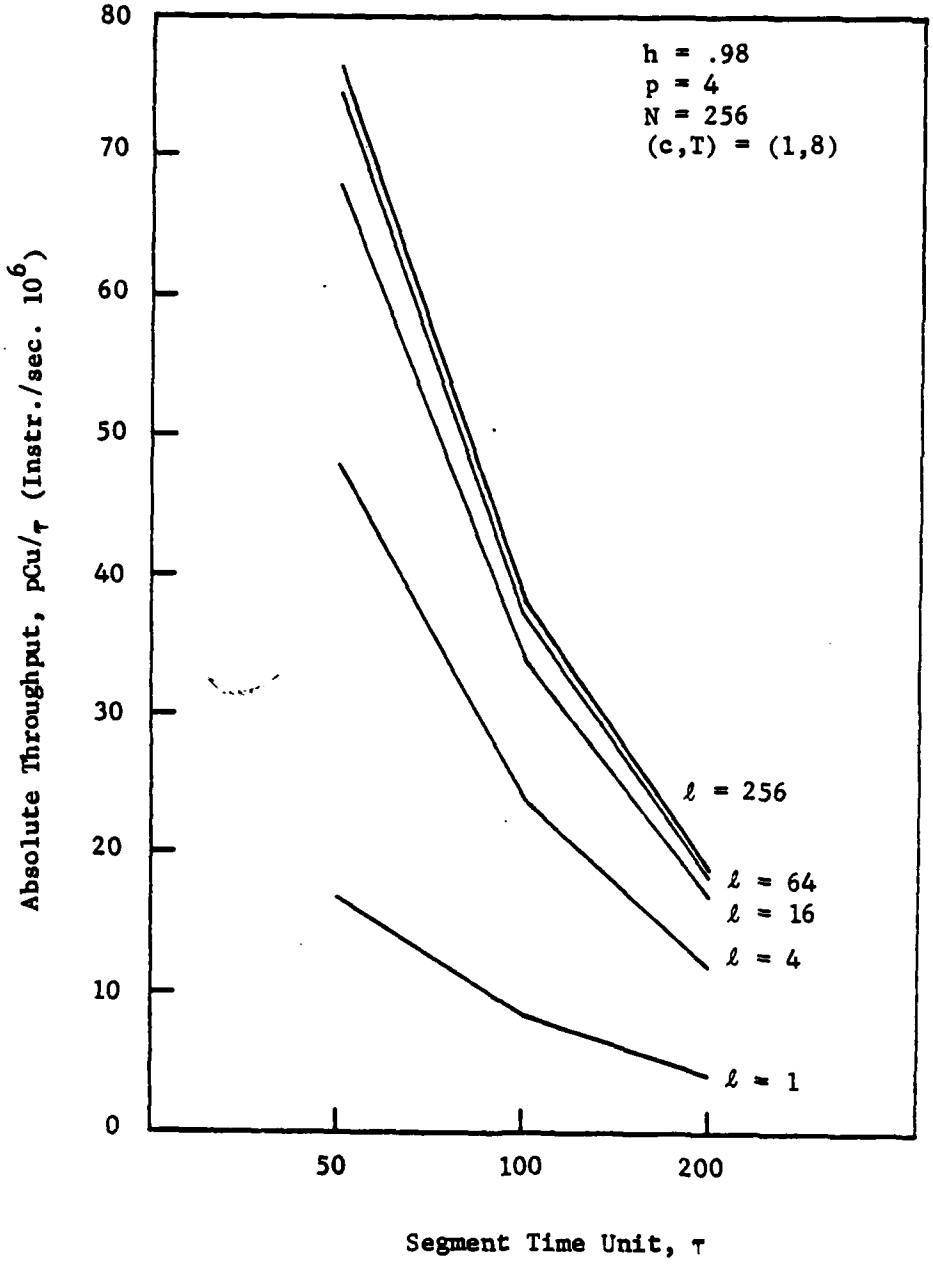


Figure 4.9.3 Effect of processor and memories speed on throughout.

configurations. In all configurations, an increase in processor speed increases the throughput proportionally. In this case, a change in the processor speed requires a similar change in memory speed for all levels of memory hierarchy.

4.10 Effect of Miss Penalties on Performance

The effects of cache access conflicts on performance for various parameters with a given hit ratio have been discussed in the previous sections. We have also discussed the effect of access conflict due to block transfer operation on system performance. In this section, the effect of miss penalties on performance is discussed. To highlight the effect of miss penalties on performance, the cache access conflict should be reduced to a minimal level. Therefore, only the case of $\ell \gg p$ is considered here. Note that the cache memory cycle, c, has an insignificant effect on performance for both models in this region.

For $\ell \gg p$, $(1-P_1) \approx P_A \approx 1$. Hence, $Cu \approx 1/[1+(1-h)T']$ for both model A and model B. The performance then depends on the product of (1-h) and T' as shown in figure 4.10.1. As an illustration, consider the case of (1-h)=0.1 and T=32, which results in a performance of Cu=0.54. Note that a constant T may be held as (1-h) varies by varying the cache capacities or the set sizes. Therefore doubling (1-h), in effect, requires T to be halved in order to keep constant performance for $\ell \gg p$. Figure 4.10.1 shows that Cu=0.7 for (1-h)=0.1 and T=16 and for (1-h)=0.05 and T=32. However, if T is primarily determined by main memory cycle, then halving

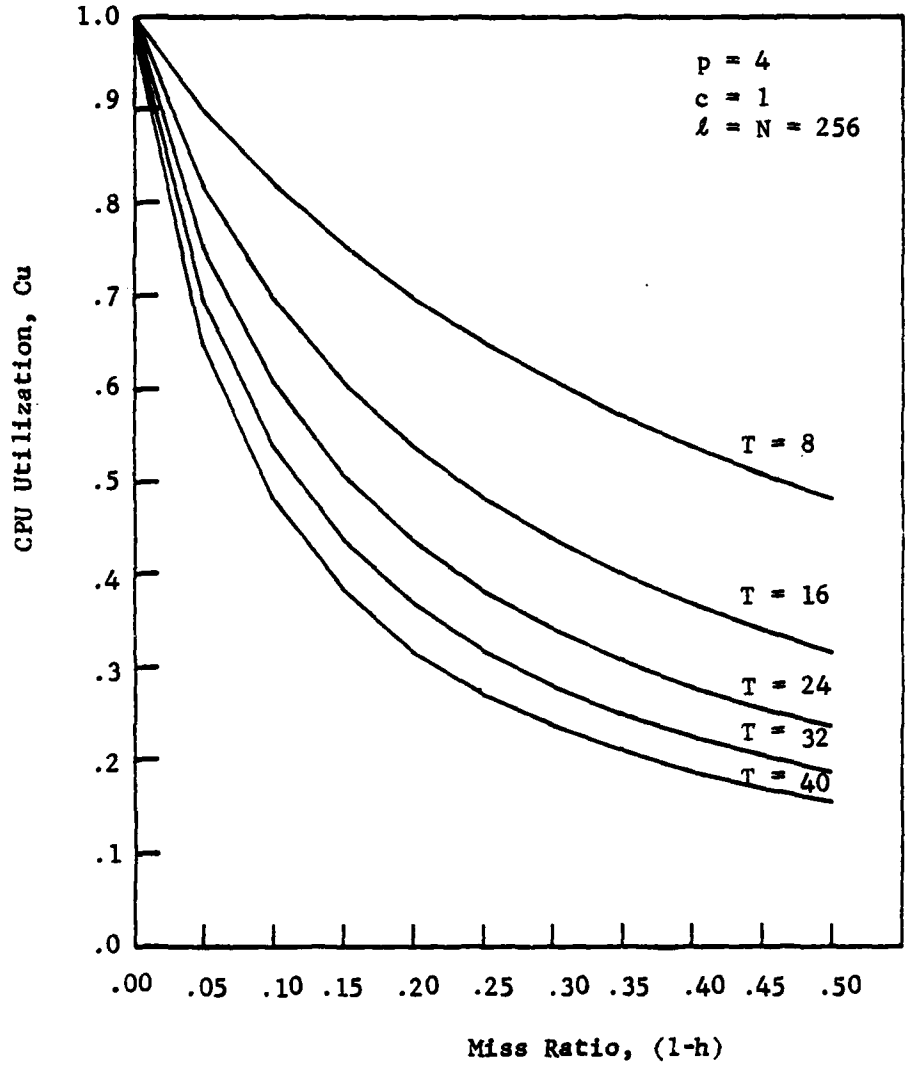


Figure 4.10.1 Effect of (1-h) on Cu for ℓ=N=256.

T may generally cost more than halving (1-h) because the size of main memory is usually much larger than that of cache memory.

Various (1-h) may also be obtained by varying the block sizes as discussed in section 4.2. In general, the block transfer time, T, may be expressed as a linear function of the block size. Let M_c and B_s represent the main memory cycle and the block size respectively. The block transfer time, T, can be expressed approximately as $T=M_c + kB_s$, where k is a constant which specifies the main memory bandwidth. As an example of selecting the block size for a 1 K cache capacity, consider figure 4.2.2, which shows miss ratios of 0.060, 0.045, 0.043, 0.049 and 0.058 for the block sizes of 2, 4, 8, 16 and 32 respectively. If M_c, k and s are assumed to be 5, 1 and 4 respectively, then miss penalties, $(1-h)T^n$, of 0.120, 0.135, 0.172, 0.294 and 0.580 result for block sizes of 2, 4, 8, 16 and 32, respectively. Therefore, a smallest block size of 2, instead of the block size corresponding to the minimum value of miss ratio, gives the highest performance in this case. Clearly, small block sizes are preferred as the constant, k, increases. For large M_c and small k, the effect of block size on performance is less significant if (1-h) is also small.

For a given cost, optimal memory hierarchies may be obtained by properly choosing (1-h) and T. Chow [64] studied the optimization of storage hierarchies based on the assumptions that the miss ratio function and the device technology cost function are representable by power functions. However, our experiments show that the shared cache miss ratio function, i.e., figure 4.2.1, cannot be fitted by a power function.

Welch [65] did a similar study by only assuming the cost function to be a power function. Welch's model is interesting here since as with our model he allows the hit ratio to be specified arbitrarily. For our two-level memory hierarchy, Welch's memory balance equation becomes

$$\frac{(1-h)T''}{T_{ave}} = \frac{B}{S}$$

where $(1-h)T''$ is the cache miss penalty, T_{ave} is the average access time of system, S is the total cost of all memory levels and B is the cost of main memory. Note that only one of the two parameters, h and T", can be varied at a time. Therefore, for fixed h, the main memory should be speeded up if the ratio of main memory cost to system cost is larger than the ratio of main memory delay to system delay. Equivalently, the main memory shoud be slowed down if the main memory results in a higher proportion of system cost than its proportion of system delay. Similarly, for fixed main memory speed, i.e. fixed T" in Welch's model, more money should be invested in cache to enhance h if the main memory results in a higher proportion of system cost than its proportion of system delay. On the other hand, the hit ratio should be reduced if the main memory absorbs a smaller proportion of system cost than its proportion of system delay. Hence, Welch's result may be helpful to determine the tradeoffs between h and T if the main memory size is known for our models.

However, his model does not consider the relationship between T" and c. It is not necessary to invest money in main memory to speed up T", T" can be reduced by spending money in cache to speed up c. Note also that

the hit ratio function is only a function of the cache capacity in his model. However, various hit ratios may be obtained by varying the block sizes and the set sizes for a given cache capacity.

Therefore, care must be taken in applying Welch's result directly to our models. For a required level of performance, the tradeoffs between h and T^m are determined by the system cost. Once again, knowledge about the cost variances for different cache capacities, main memory capacities, device technologies, block sizes and set sizes are needed.

In summary, for $\ell >> p$, the performance is only sensitive to the cache miss penalty, $(1-h)T^m$, for both model A and model B. However, the tradeoffs between h and T^m involve many cost functions. In this region, the shared-cache system may perform better than the private-cache system if shared cache results in a smaller miss ratio.

4.11 Load Through versus Nonload-Through

In general, there are three ways to handle a cache miss on read: (1) load through, (2) nonload-through with processors resubmitting the same request every cycle during the block transfer time, and (3) nonload-through with processors not making any request during the block transfer time. Nonload-through was assumed for both the analytic models and simulation experiments discussed before. Also, the request rate was assumed to be one for all models and experiments and blocked requests were handled by resubmitting them as new requests in the analytical

models (the same requests are resubmitted in the simultaion experiments) one instruction cycle later until they are satisfied. Therefore, case 2 mentioned above has been simulated and modeled. The analytic models developed in chapter 3 can easily be extended to case 3 by using lemma 3.4.1 and theorem 3.4.2 to modify the request rate. If the load through capability is provided, processors do not have to resubmit a blocked request due to cache miss since it is satisfied when the miss is accepted. In this case, only those blocked requests caused by cache access conflicts have to be resubmitted, therefore, the processor request rate is less than one and the multiple access line collisions are reduced. The miss penalty may also be reduced for load through because the processor waiting time for obtaining miss data is usually less than the block transfer time. In this section, the effects of load through on performance will be discussed based on model A. This discussion can be easily extended for model B with load through.

Let W denote the processor waiting time, measured in STUs, for obtaining the miss data after a cache miss occurs. Usually, this waiting time is approximately equal to the main memory cycle. By an argument similar to that used in the proof of theorem 3.4.2, it is obvious that each request will extend to $1/P_A$ requests due to cache access conflicts and each cache miss causes no request for $\lceil W/s \rceil$ instruction cycles. Therefore, the actual request rate seen by the shared cache for model A with load through is given as

$$\alpha = \frac{1/P_A}{1/P_A + (1-h)\left\lceil \frac{W}{s} \right\rceil}$$

Assume that the block transfer time, T, is not affected by the load through. Then, the probability of acceptance, P_A, is evaluated as before except that the probability of a request being rejected due to multiple access line collision, P_1, is given by lemma 3.4.1, i.e. $1 - [1 - (1 - \alpha/\ell)^p]\ell/\alpha$ p. instead of theorem 3.2.1.1, i.e., $1 - [1 - (1 - 1/\ell)^p]\ell/p$. Thus P_1 can be expressed in terms of P_A by substituting for α the expression given above. Using this P_1 in the equation for P_A, we have an equation in which P_A is the only unknown. This equation can be numerically solved using standard iterative techniques. A suitable initial value for P_A is obtained by setting $\alpha = 1$. The performance, CPU utilization is then obtained as

$$Cu = \frac{1}{\frac{1}{P_A} + (1-h)\left\lceil \frac{W}{s} \right\rceil}$$

Figure 4.11.1 illustrates the performance difference between load through and nonload-through for a fixed processor waiting time, W=4. In this case, different values of T can be explained as the result due to the variations in main memory bandwidth or block size if fixed W implies a fixed main memory cycle. It can be seen that load through performs significantly better than nonload-through, especially for small h and large T. Figure 4.11.2 shows the performance variations for load through

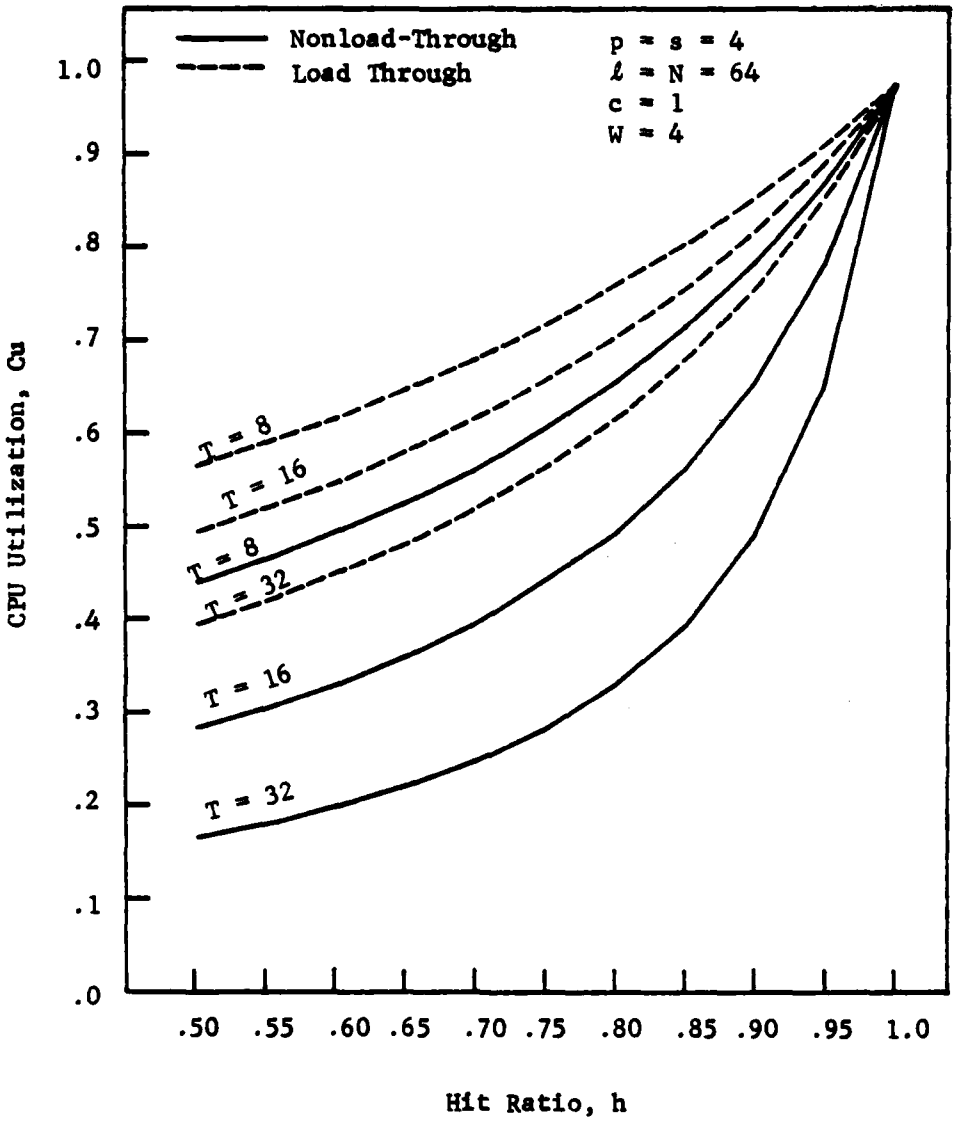


Figure 4.11.1 Performance comparison between load through and nonload-through for a fixed W=4

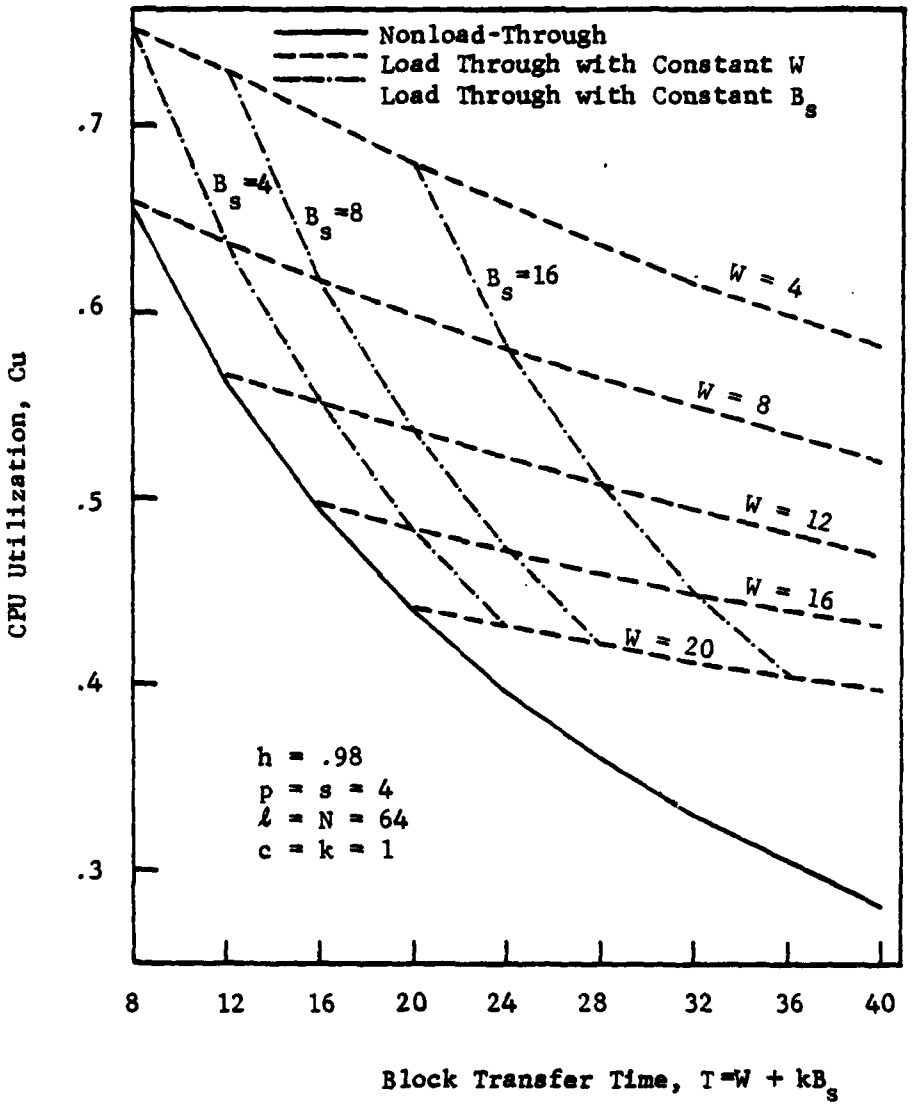


Figure 4.11.2 Performance comparison between load through and nonload-through for various W's and B_s's.

due to various waiting times (W curves) for a given hit ratio, h=0.8. Clearly, the larger the difference between W and T, the larger the performance improvement from load through.

As stated in section 4.10, the block transfer time, T, can usually be expressed as a linear combination of main memory cycle, M_c, and block size, B_s, i.e. $T=M_c + kB_s$. Since the processor waiting time, W, for cache miss is usually approximately M_c, variations in M_c will cause both T and W to change simultaneously. If $W=M_c$ and $k=1$, then the difference between T and W is the block size. In this case, load through performs significantly better than nonload through for large block sizes. The B_s curves in figure 4.11.2 illustrate this effect. The B_s curves are constructed by selecting $T=B_s+W$ points on each W curve and connecting points over all W curves with constant B_s.

In summary, load through performs significantly better than nonload-through for small h and a large difference between W and T. Note that load through reduces the waiting time required to obtain the data which cause misses. In order to access the next data, the processor may still have to wait until the block transfer operation is completed. This situation is due to program localities which may cause the next data accessed to be in the same block as the currently referenced data. Since the addresses of the requests are assumed to be independent and random, this effect has not been modeled. However, this effect is reduced when the difference between W and T is small.

4.12 Comparisons Between Shared Cache and Private Cache

The performance differences for various parameters have been discussed in the previous sections for shared-cache systems. In this section, performance comparisons between shared cache (model A) and private cache for several different organizations are discussed. The possible overhead to handle the multicopy of data problem for private cache is not considered here. In figure 4.2.2, miss ratios of 0.015 and 0.03 are shown for shared cache with write through and private cache with write back respectively for a cache capacity of 2048, block size=8, set size=8, and sp=4. These miss ratios are used as given parameters to investigate the performance difference between shared cache (model A) and private cache for various system organizations. Let $(1-h_s)$ and $(1-h_p)$ denote the miss ratios for shared cache and private cache, respectively. Also let ℓ denote either the number of lines in cache memory for shared cache or the number of lines in main memory for private cache systems.

For the first case, system organizations of multiprocessors with four nonpipelined processors, i.e., (s,p) = (1,4), for shared cache and private cache are illustrated in figure 4.12.1(a) and figure 4.12.1(b), respectively. The performance prediction for private cache can be obtained by using theorems 3.4.1, 3.4.2, and 3.4.3. The analytic equations to predict the performance for shared cache are given as

$$P_A = \frac{\ell(1-P_1)}{\ell + pT''(1-P_1)(1-h_s)}$$

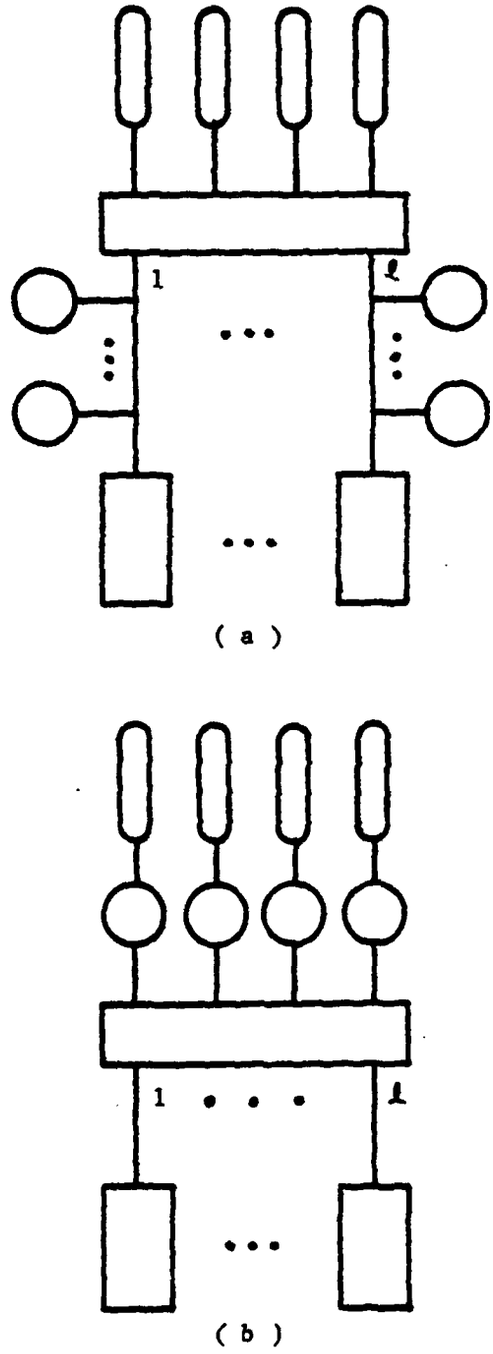


Figure 4.12.1 Multiprocessor systems with nonpipelined processors for (a) shared cache and (b) private cache.

$$Cu = \frac{1}{\frac{1}{P_A} + (1-h_s)T''} ,$$

where T" is the block transfer time relative to the processor cycle.

Since processors make one cache memory request every instruction (or processor) cycle and the cache cycle is assumed to be less than one instruction cycle, the probability of acceptance, P_A, given above is obtained from Appendix A for c=1 and T is replaced by T". Figure 4.12.2 shows the performance comparison between shared cache and private cache for this case. Note that the solid line curves are the analytic results for shared cache with various ℓ and the dotted line curve is the analytical result for private cache with ℓ =4. The topmost solid line curve is the absolute upper bound, i.e., for P_A=1, performance for shared cache. However, for private cache with ℓ =4, the main memory access interference is negligible, i.e. $P_{AM} \approx 1$, in this example. Therefore, the upper bound for shared cache is higher than that for private cache because $(1-h_s)$ is smaller than $(1-h_p)$. Although the performance of shared cache is much worse than that of private cache for ℓ =4, shared cache may perform better than private cache, especially for large T, if a sufficiently large ℓ is used for shared cache. For example, shared cache with ℓ =16 performs better than private cache with ℓ =4 for T" 8.

For the second case, a pipelined processor with four segments is considered. Figures 4.12.3(a) and (b) show the system organizations for shared cache and private cache, respectively. Note that for p=1, no

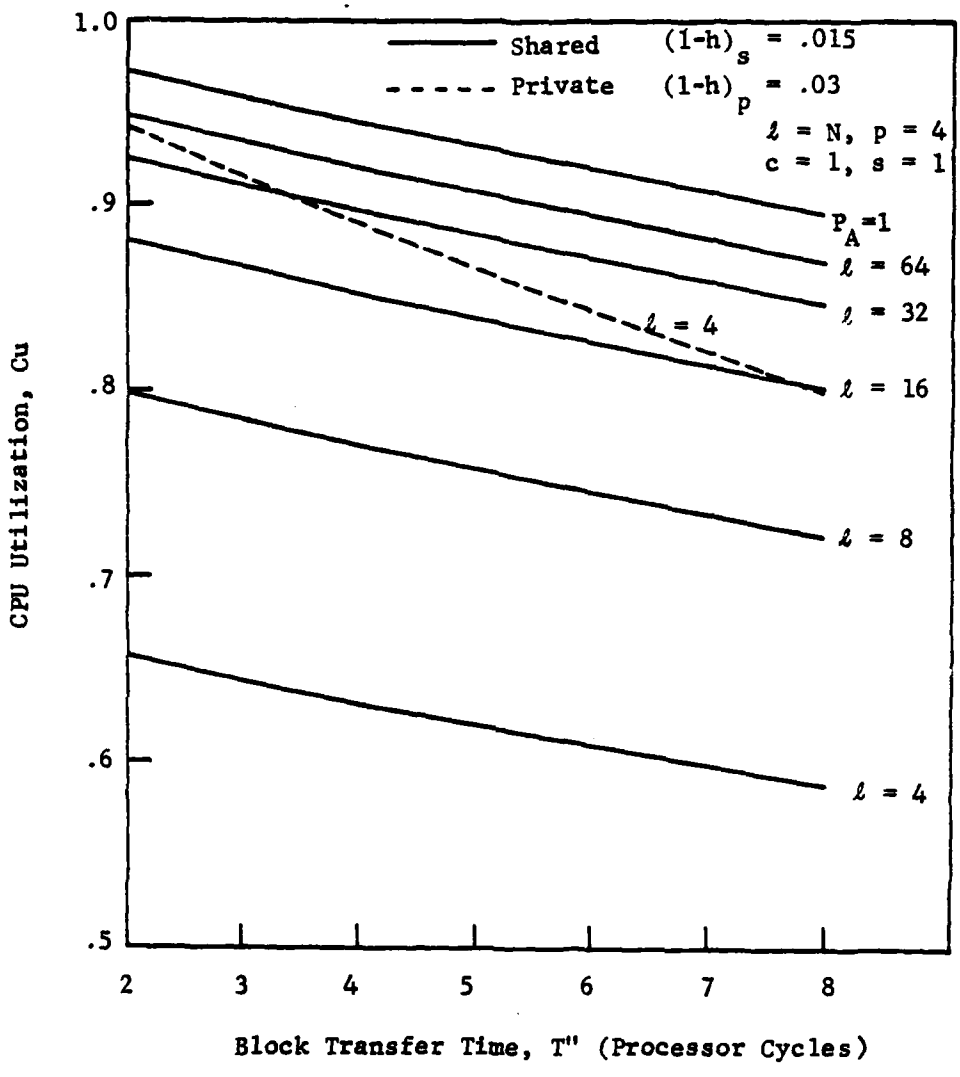


Figure 4.12.2 Performance comparison between shared cache and private cache for nonpipelined multiprocessor systems.

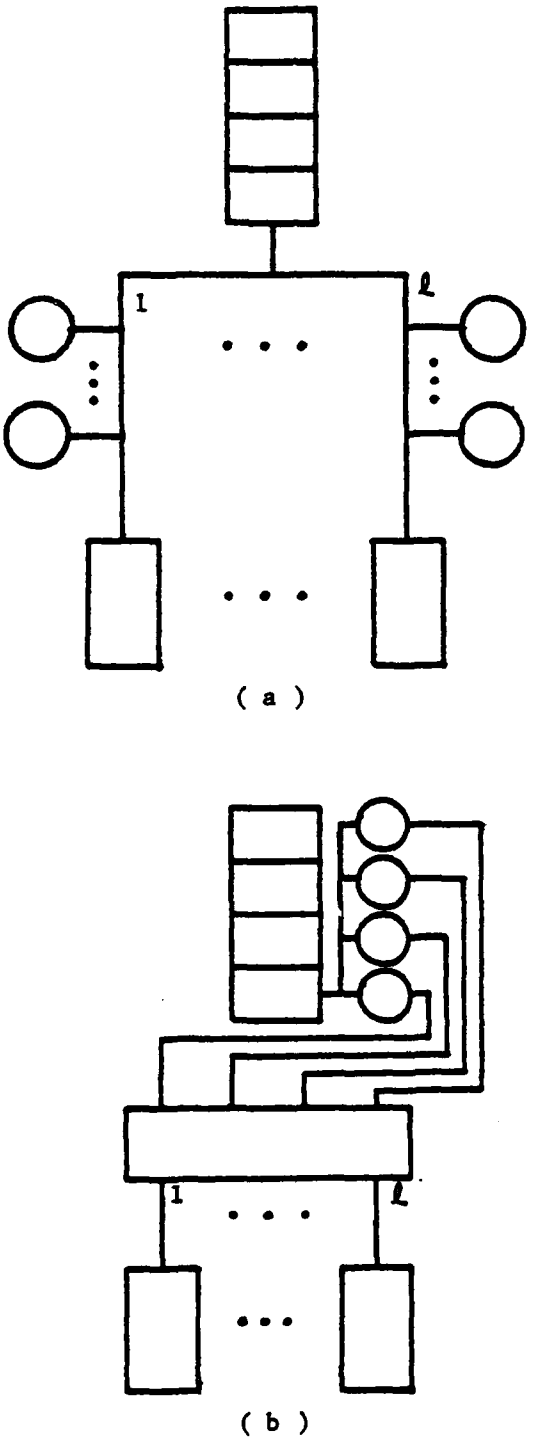


Figure 4.12.3 Single pipelined processor systems for (a) shared cache and (b) private cache.

crossbar is needed for shared cache systems. For private cache systems, there are four cache modules, each associated with one process, connected through a time-multiplexed bus to the processor and there is a crossbar switch between the cache modules and the main memory modules. For shared cache, the analytic equations used to predict performance can be obtained by combining theorem 4.3.3 and a proper equation in Appendix A for a specific c. However, for private cache, the performance can be predicted by a direct extension of section 3.4.

In the derivation of theorem 3.4.1, it was shown that the probability of acceptance for a single resource requested by a pipelined processor is $1/[\alpha(T-1)+1]$, where α and T are the request rate and resource cycle respectively. Briggs [42] and Emer [60] have derived similar results. This result is then the probability of acceptance of a main memory request for the private cache shown in figure 4.12.3(b). The resource cycle, T, is the block transfer time in this case. However, the request rate α should be the actual request rate seen by the cache. This rate can be obtained by applying theorem 4.3.2. Hence the analytic equations for private cache are the following two equations together with the CPU utilization equation obtained from theorem 4.3.3:

$$P_{AM} = \frac{1}{\alpha(T-1)+1}$$

$$\alpha = \frac{1}{1+T'' P_{AM} + (\frac{\ell}{1-h} - 1)P_{AM}}$$

$$\text{and } C_u = \frac{1}{1+(1-h)[\frac{1}{P_{AM}} -1+T"]}$$

Figure 4.12.4 illustrates the performance comparison between shared cache and private cache for ℓ =N. Note that the cache cycle time, c, for private cache is not explicitly shown because it does not pose any limitation on performance as long as the cache cycle meets the deadline required by the processor. Again, the topmost line shows the absolute upper bound, i.e. P_A=1, performance for shared cache. As can be seen, shared cache always performs better than private cache for c=1. For large ℓ , a significant improvement in performance results for shared cache, especially for large T. A performance comparison between shared cache and private cache for a fixed ℓ , i.e. 4, is shown in figure 4.12.5. Note that for shared cache and a given ℓ , the performance for a system with cache cycle time k is asymptotically bounded by the performance for the system with cache cycle time k-1 and N= ℓ as N approaches infinity. This asymptotic behavior of increasing N can be seen by examining the equations listed in Appendix A. Figure 4.12.5 shows that shared cache with c=2 and private cache may result in comparable performance if a large N is feasible for shared cache.

As a result of improvements in fabrication technology, many LSI chips, such as microprocessors, now consist of multiple complex subsystems. However, one of the main cost factors and fabrication difficulties is pin count. For the third case considered below, it is

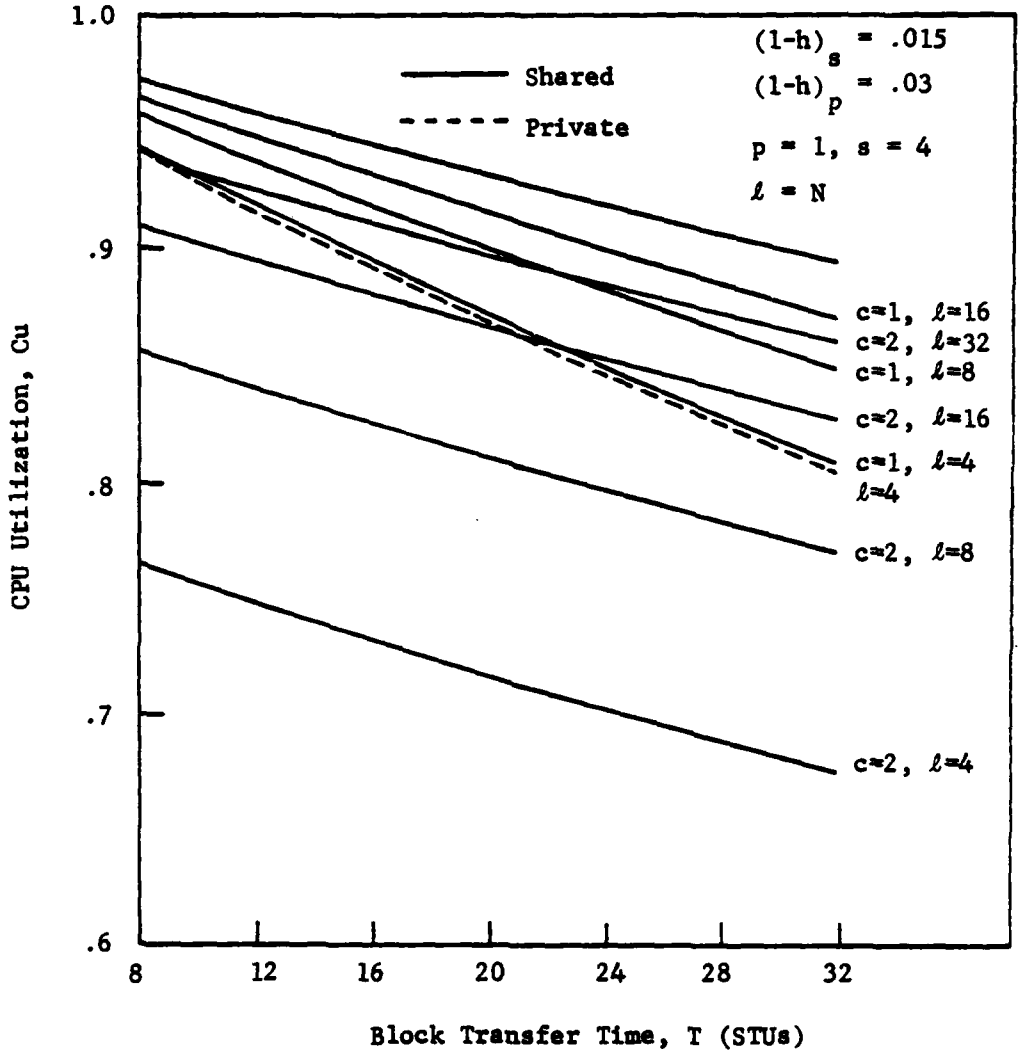


Figure 4.12.4 Performance comparison between shared cache and private cache for single pipelined processor with ℓ=N and s=4.

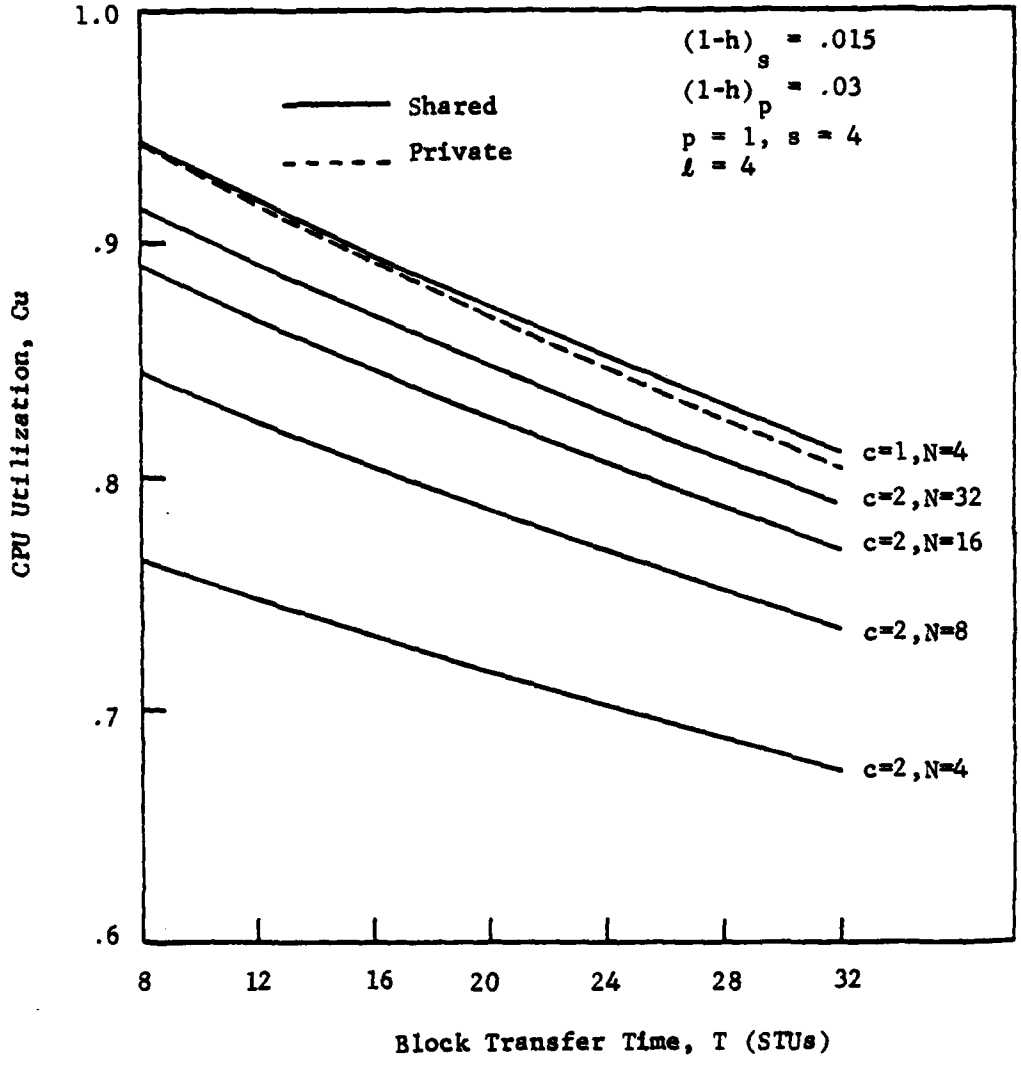


Figure 4.12.5 Performance comparison between shared cache and private cache for single pipelined processor with ℓ=4 and s=4.

assumed that a pipelined processor and its cache memory modules are integrated in a single chip for both shared cache and private cache. In order to reduce the number of pins, a time-multiplexed bus connected between cache modules and main memory modules is used for both systems. Figure 4.12.6(a) and figure 4.12.6(b) illustrate the system organizations for shared cache and private cache, respectively. Obviously, the organization shown in figure 4.12.6(b) is a special case, i.e. $\ell = 1$, of that shown in figure 4.12.3(b). The analytic equations used to predict performance for the system shown in figure 4.12.3(b) can be directly applied to predict the performance for the private cache system in this case.

For the shared cache in this case, it is assumed that the line is busy for the period starting from detecting a cache miss on this line until the block transfer operation is complete. Therefore, the effective block transfer time, denoted as T' actually includes the waiting times for both main memory request due to access conflict and block transfer operation. The change of main memory request rate, due to main memory bus contention, changes the effective block transfer time. Thereby, changes also occur in the probabilities of acceptance, P_A and P_{AM} for a cache request and for a main memory request, respectively. The analytic solution is not trivial but can be found for a specific c. As an illustration, consider c=1, the probabilities of acceptance for both a cache request and a main memory request are given as $P_A = \ell/[\ \ell+(1-h)T'\ ]$ and $P_{AM} = 1/[\ \alpha(T-1)+1\ ]$ respectively, where T' is the actual (or effective) block transfer time and α is the actual main memory request

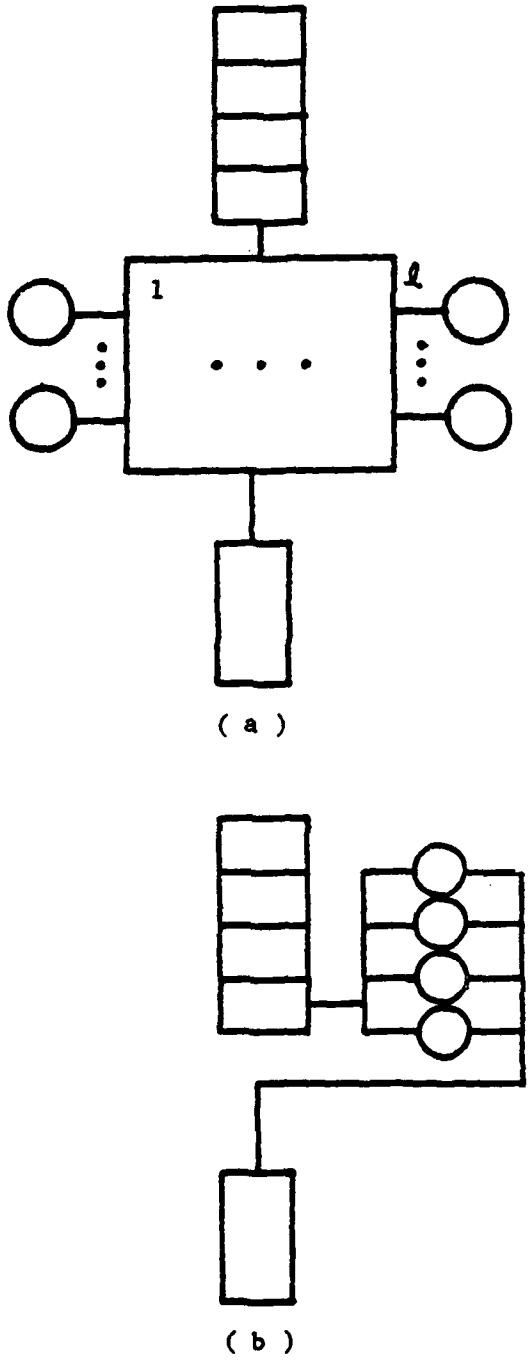


Figure 4.12.6 Single pipelined processor with time-multiplexed main memory bus for (a) shared cache and (b) private cache.

rate. Note that T′ can be expressed as, $(1/P_{AM} - 1)s + T$ or alternately as, $\alpha(T - 1)s + T$. Hence $P_A = \ell / \{ \ell + (1-h)[\alpha(T-1)s + T]\}$. Note also that the main memory request rate, α, is $(1-h)P_A$ because only an accepted cache request which results in a miss can make a main memory request. Then the solution of the following equations gives us the probability of acceptance P_A. These equations can be solved numerically using well known algorithms. A suitable starting value for P_A, useful in iterative solution methods, is $\ell/[\ \ell + T(1-h)]$.

$$P_A = \frac{\ell}{\ell+(1-h)[\alpha(T-1)s+T]}$$

$$P_{AM} = \frac{1}{\alpha(T-1)+T}$$

$$\alpha = \frac{1}{1+T'' P_{AM}+(\frac{1}{(1-h)P_A} -1)P_{AM}}.$$

Using the value of P_A and α from the above equations, we can obtain the CPU utilization for cache cycle c=1 as follows.

$$Cu = \frac{1}{\frac{1}{P_A} + (1-h)\left\lceil \frac{T'}{s} \right\rceil} ,$$

where $T' = \alpha(T-1)s + T$.

The corresponding P_A chosen from Appendix A is used in the above equations with T replaced by T′ for each c. Figure 4.12.7 and figure

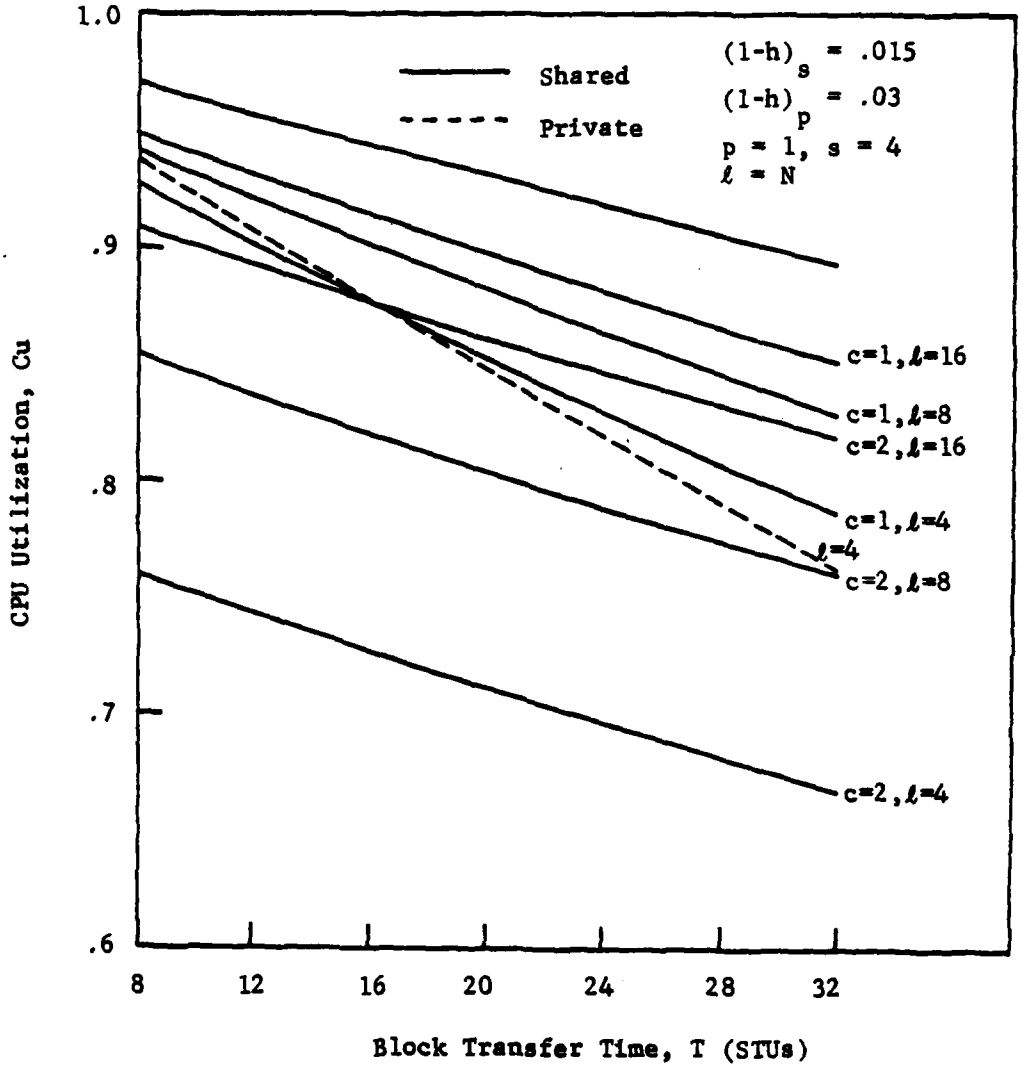


Figure 4.12.7 Performance comparison between shared cache and private cache for p=1, s=4, ℓ=N and a single time-multiplexed main memory bus.

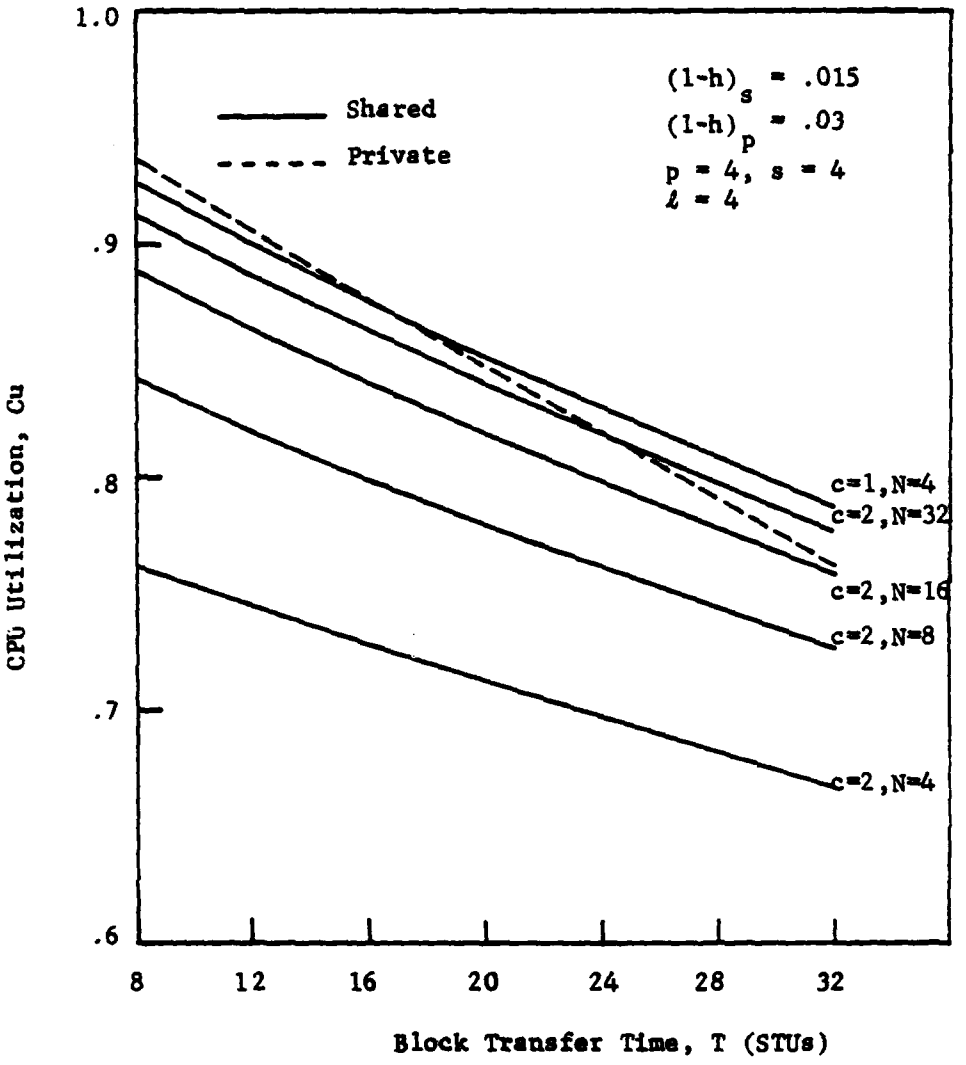


Figure 4.12.8 Performance comparison between shared cache and private cache for p=1, s=4, ℓ=4 and a single time-multiplexed main memory bus.

4.12.8 illustrate the performance comparisons between shared cache and private cache, both with shared main memory bus, for ℓ = N and ℓ = 4 respectively. As can be seen from figures 4.12.4, 4.12.5, 4.12.7 and 4.12.8, for private cache, the performance degradation due to the shared main memory bus is significant when T is large. However, for shared cache this performance degradation is less significant because $(1-h_s)$ is small.

In general, shared cache may perform better than private cache if $(1-h_s)$ is smaller then $(1-h_p)$ and ℓ is sufficiently large. Shared cache is especially suitable for single pipelined processors, as in the second and the third cases discussed in this section, because high performance can easily be obtained and no crossbar is required.

CHAPTER 5

CONCLUSIONS

5.1 Summary of Results

This research develops a simple and flexible system organization for parallel-pipelined processors with a shared two-level memory hierarchy and investigates the effect of hit ratio on system performance for various workloads and cache component parameters as well as the effect of cache memory access interference on system performance for a variety of cache memory configurations, cycle characteristics, and processor speeds. The multicopy of data problems in conventional multiprocessors with private caches are totally eliminated by the architectural approach of sharing the caches. The shared cache hit ratio function has been investigated for a range of component parameters and several combinations of four real program traces and two operating environments, that is Independent Instruction-stream, Independent Data-stream (IIID) and Shared Instruction-stream, Independent Data-stream (SIID) were used for experimental evaluation by means of simulation. The shared-cache systems have been studied for two kinds of organizations, namely shared cache with an implicit lookup table and shared cache with explicit lookup tables. For each shared cache organization, the effect of cache access interference on system performance has been investigated for three cache

cycle times c=1, 2, and 3. The complexity of the Markov state diagram grows exponentially with c. So far, we do not have a general solution of performance for arbitrary c.

In chapter two, the L-M memory organization was reviewed and cache memory management strategies were discussed. A set associative mapping mechanism with a modified LRU replacement algorithm was assumed for shared-cache systems. The write-through with buffering updating scheme and the no-write allocation strategy were used in a shared-cache multiple-stream system. However, a flagged register swap algorithm and the write allocation strategy were used for multiprocessor systems with private cache memories. Furthermore, a shared-cache memory interleaved by sets was introduced.

In chapter three, we developed analytical models for both shared cache with an implicit lookup table and shared cache with explicit lookup tables. In addition, an analytic model used to evaluate the main memory interference for a multiprocessor system with private cache memories was also developed. These models were oriented toward deriving the probability of acceptance, $P_A(c,T,p)$, of a request. The performance measurement, CPU utilization, was obtained for each model based on $P_A(c,T,p)$. The hit ratio was left unevaluated and assumed to be a specified parameter in developing the analytical models. Since a general expression for $P_A(c,T,p)$ is not known for arbitrary cycle time, c, we have obtained upper and lower bounds on $P_A(c,T,p)$ for the implicit lookup table model, but were unable to do so for the explicit lookup table model.

In chapter four, simulation experiments have been used to investigate the effect of various workloads and component parameters on hit ratio and to validate the analytic models. We demonstrated that the simulation results were not significantly different from the analytic predictions for reasonably high performance systems. This justifies the assumption that the discarding of rejected requests, for analytical purposes, does not necessitate a significant deviation of the analytic model from reality for reasonably high performance systems. We also demonstrated that the hit ratio deviations due to cache memory interference are insignificant. This justifies the assumption that the hit ratio is constant with respect to changes in access conflict.

Experiments showed that shared cache is more sensitive to set sizes than private cache. We observed in our simulation experiments that write-through is always superior to write-back. In most cases, dynamic space sharing was better than fixed space allocation for the IIID operating environment. The most significant improvements in miss ratio by shared cache over private cache occurred for the SIID operating environment.

We also investigated the effect of memory interference for a variety of parameters on system performance. Since the analytic predictions for shared cache with an implicit lookup table and explicit lookup tables are almost the same for the range of parameters we studied, the discussion was restricted only to shared cache with an implicit lookup table. We found that for a very large number of cache memory modules, N, the effect of cache cycle, c, is insignificant for shared cache with an implicit

lookup table.

There is generally less payoff to increasing N for large ℓ and small p, and for small ℓ and large p. The most significant payoff to increasing N occurs when ℓ is close to p.

We showed that systems result in very poor performance for $\ell < p$. For $\ell = p$, the performance is sensitive to small variations in ℓ, p, T, and N. In order to obtain reasonable performance, $\ell > p$ is necessary.

The effect of cycle characteristics on performance is small when ℓ and N are sufficiently large. We have shown that for small ℓ and N, the effect of cache cycle, c, is significant.

The processor speed is another important factor that determines the system throughput. For both a higher request rate and a uniformly faster processor/memory, i.e. fixed (c,T), assumptions, an increase in the processor speed increases the throughput. For a constant request rate assumption, an increase in the processor speed decreases the throughput slightly for large ℓ but increases the throughput for small .

We have shown, for sufficiently large ℓ, that system performance is critically dependent on the miss penalty, $(1-h)T^{m}$. One example shows that maximum performance may not be produced for the block size which corresponds to the minimum value of miss ratio.

A simple model has been developed for processors with load through capability. We illustrated that load through is significantly better than nonload-through for small h and a large difference between T and the

main memory cycle.

Performance comparisons between shared cache and private cache were carried out for several organizations. We showed that shared cache with write-through is especially suitable for single pipelined processor systems.

In general, since shared space could be equally divided amoung the processes, shared cache under an effective management policy should yield a hit ratio at least as high as that for private cache. However, it was shown in section 4.3 that shared cache might result in a higher miss ratio for an LRU replacement policy per set. More research on dynamic space sharing and the interaction between streams is required to derive an effective management policy for shared cache.

If shared cache gives a higher hit ratio than private cache, then shared cache results in higher system performance for those configurations that keep the access conflict at low levels. Note that the overhead caused by handling the multicopy of data problem has not been considered for private cache systems. The performance predictions for private cache are thus optimistic.

If both shared cache and private cache have the same cache organization, then private cache is more expensive than shared cache because a "store controller" and a "central directory" [16] have to be provided in order to solve the multicopy of data problem. This cost difference can be invested in shared cache to reduce access conflict and enhance system performance. In addition to possible higher performance,

shared cache systems have the following advantages over private cache systems:

(1) no multicopy of data problem,

(2) interprocessor communication can be implemented in the cache, and

(3) design is simpler.

5.2 Suggestions for Further Research

As mentioned before, in order to derive an effective management policy for shared cache, more research on policies for management of dynamic space sharing should be done. Shared-cache systems with separate caches for instructions and data may be interesting. In practice, space allocation for data and instructions in such computer systems may be complex.

The performance evaluation of pipelined processors with both load through and instruction prefetch capabilities is important. Both load through and prefetch can be carried out simultaneously if the bandwidth for load through is greater than one. In this case, not only the read-miss instructions but also the sequential instructions following those read-miss instructions will be loaded from main memory directly to processors.

A possible extension of this thesis may be directed towards

developing a model for general memory hierarchies. It may be possible to characterize the access time to a memory level in terms of the access times to all the higher level memories. A hierarchical model would be very useful because any later change of technology for some memory level would require that only the model for that level be modified.

Finally, the effective buffer sizes for both write-through and write-back should be investigated. Software developments, such as microprogramming and resource allocation, for shared cache systems should also be studied.

APPENDIX A

A Summary for Shared Cache with An Implicit Lookup Table

1. m = 1, $$P_A = \frac{\ell q}{\ell p + pq(c-1) + Tpq(1-h)}$$

2. (c, T) = (1, T), $$P_A = \frac{\ell q}{\ell p + Tpq(1-h)}$$

3. (c, T) = (2, T), $$P_A = \frac{\ell qN}{\ell pN + Npq(T+1)(1-h) + pq\ell h}$$

4. (c, T) = (3, T), $$P_A = \frac{\ell N(1-P_1)(N+qh)}{\Delta}$$

where $\Delta = (\ell+2q+Tq-Thq-2hq)N^2 + (3qh\ell-h^2q^2+hq^2+Thq^2-Th^2q^2)N + \ell h^2q + h\ell q^2$,

$q = 1-(1-1/\ell)^p$, and

$(1-p_1) = [1-(1-1/\ell)^p]\ell/p$.

5. CPU Utilization, $$Cu = \frac{1}{1/P_A + (1-h)T''}$$

APPENDIX B

A Summary for Shared Cache with Explicit Lookup Tables

1. $m = 1$,

$$P_{Ah} = P_{Am} = \frac{\ell q}{\ell p + phq(c-1) + pq(T-1)(1-h_d)}$$

$$h = h_d$$

2. $(c, T) = (1, T)$,

$$P_{Ah} = P_{Am} = \frac{\ell q}{\ell p + pq(T-1)(1-h_d)}$$

$$h = h_d$$

3. $(c, T) = (2, T)$,

$$P_{Ah} = \frac{\ell qN}{\ell Np + pq(T-1)(1-h_d)[N - q(m-1)h_d] + pqh_d\ell}$$

$$P_{Am} = \frac{\ell N - \ell q(m-1)h_d}{\ell N + q(T-1)(1-h_d)[N - q(m-1)h_d] + qh_d\ell}$$

$$h = \frac{(1-h_d)(N-Nqh_d + \ell qh_d)}{Nh_d}$$

4. $(c, T) = (3, T)$

$$P_{Ah} = \frac{\ell N(1-P_1)(N+hq)}{\Delta}$$

$$P_{Am} = \frac{\ell Y_1 Y_2}{\Delta}$$

$$h = \frac{(1-h_d)(N - Nqh_d + \ell qh_d)(N - Nqh_d + 2\ell qh_d)}{Nh_d(N + \ell qh_d)}$$

where $\Delta = \ell N(N + h_d q) + q(T-1)(1-h_d)Y_1Y_2 + 2qh_d\ell(N + h_d q)$,

$Y_1 = (N-mh_d q + h_d q)$, $Y_2 = (N - mh_d q + 2h_d q)$,

$q = 1-(1-1/\ell)^p$ and $(1-P_1) = [1-(1-1/\ell)^p]\ell/p$.

* h : static hit ratio; h_d : dynamic hit ratio.

APPENDIX B
(continued)

5. CPU Utilization, $$Cu = \frac{1}{h/P_{Ah} + (1-h)(1/P_{Am}+T'')}$$

REFERENCES

(1) Chen, T. C., "Parallelism, Pipelining, and Computer Efficiency", Computer Design, pp. 365-372, January 1971.

(2) Bell, C. G., and Wulf, W. A., "C.mmp-A Multiminiprocessors," AFIPS Proc. FJCC, Vol. 41, Part II, pp. 765-777, 1972.

(3) Davidson, E. S., "A Multiple Stream Microprocessor Prototype Systems: AMP-1", The 7th Annual Symposium on Computer Architecture pp. 9-16, May 1980.

(4) Flynn, M. J., "Very High-Speed Computing Systems", Proc. of the IEEE, Vol. 54, No. 12, pp. 1901-1909, December 1966.

(5) Barnes, G. H., et al., "The ILLIAC IV Computer", IEEE Trans. Comput., Vol. C-17, No. 8, pp. 746-757, August 1968.

(6) Batcher, K. E., "STARAN Parallel Proessor System Hardware," AFIPS Proc. NCC, Vol. 43, pp. 405-410, 1974.

(7) Crane, B. A., et al., "PEPE Computer Architecture", IEEE COMPCON, pp. 57-60, 1972.

(8) Anderson, D. W., et al., "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling", IBM J. of Res. and Dev., pp. 8-24, January 1967.

(9) Amdahl 470 V/6 Machine Reference Manual, Amdahl Corporation, Sunnyvale, Calif., 1976.

(10) Hintz, R. G., and Tate, D. P., "Control Data STAR-100 Processor Design", Proc. COMPCON Fall 72, pp. 1-4, September 1972.

(11) Watson, W. J., "The TI ASC-A Highly Modular and Flexible Computer Architecture", AFIPS Proc. FJCC, Vol. 41, Part I, pp. 221-228, 1972.

(12) Russell, R. M., "The CRAY-1 Computer System", Commun. ACM, Vol. 21, No. 1, pp. 63-72, January 1978.

(13) Shar, L. E., and Davidson, E. S., "A Multiminiprocessor System Implemented Through Pipelining", Computer, Vol. 7, No. 2, pp. 42-51, February 1974.

(14) Larson, A. G., and Davidson, E. S., "Cost-Effective Design of Special-Purpose Processor: A Fast Fourier Transform Case Study", Proc. 11th Annual Allerton Conf. on Circuit and System Theory, pp. 547-557, October 1973.

(15) Kaminsky, W. J., and Davidson, E. S., "Developing A Multiple - Instruction - Stream Single-Chip Processor ", Computer, pp. 66-76, December 1979.

(16) Tang, C. K., "Cache System Design in the Tightly Coupled Multiprocessor System", AFIPS Proc. NCC, Vol. 45, pp. 749-753, 1976.

(17) Censier, L. M., and Feautrier, P., "A New Solution to Coherence Problems in Multicache Systems", IEEE Trans. Comput., Vol. C-27, No. 12, pp. 1112-1118, December 1978.

(18) Davidson, E. S., "Effective Control for Pipelined Computers", Proc. COMPCON Spring 75, pp. 181-184, February 1975.

(19) Strecker, W. D., "An Analysis of the Instruction Execution Rate in Certain Computer Structures", Ph.D. Thesis, Carnegie-Mellon Univ., Pittsburgh, Pa., 1970.

(20) Weller, D. L., and Davidson, E. S., "Optimal Searching Algorithms for Parallel - Pipelined Computers", Spring-Verlag Lecture Notes, No. 24, pp. 90-98, August 1975.

(21) Smith, A. J., "Sequentiality and Prefetching in Data Base Systems", ACM Trans. on Data Base Sys., pp. 223-247, September 1978.

(22) Rau, B. R., and Rossmann, G. E., "The Effect of Instruction Fetch Strategies upon the Performance of Pipelined Instruction Units", Fourth Annual Symposium on Computer Architecture, pp. 80-89, March 1977.

(23) Mattson, R. L., et al., "Evaluation Techniques for Storage Hierarchies", IBM Syst. J., pp. 78-117, No. 2, 1970.

(24) Denning, P. J., "The Working Set Model for Program Behavior", Commun. ACM, Vol. 11, No. 5, pp. 323-333, May 1968.

(25) Gschwind, H. W., and McCluskey, E. J., Design of Digital Computers, Springer - Verlag, 1975.

(26) Foster, C. C., Content Addressable Parallel Processors, Van Nostrand Reinhold, 1976.

(27) Lamb, S., "An Add-In Recognition Memory for S-100 Bus Microcomputers - Part 2: Structure and Specifications", Computer Design, pp. 162-168, September 1978.

(28) Belady, L. A., and Kuehner, C. J., "Dynamic Space - Sharing in Computer Systems", Commun. ACM, Vol. 12, No. 5, pp. 282-288, May 1969.

(29) Coffman, E. G., and Ryan, T. A., "A Study of Storage Partitioning Using a Mathematical Model of Locality", Commun. ACM, Vol. 15, No. 3, pp. 185-190, March 1972.

(30) Juan Rodriguez-Rosell, "Empirical Working Set Behavior", Commun. ACM, Vol. 16, No. 9, pp. 556-560, September 1973.

(31) Hendrik Vantilborgh, "Working Set Dynamics", Modelling and Performance Evaluation of Computer Systems, Edited by E. Gelenbe, North - Holland Pub. Co., pp. 377-387, 1976.

(32) Hellerman, H., Digital Computer System Principles, New York: McGraw-Hill, pp. 228-229, 1967.

(33) Knuth, D. E., and Rao, G. S., "Activity in Interleaved Memory," IEEE Trans. Comput., Vol. C-24, No. 9, pp. 943-944, September 1975.

(34) Burnett, G. J., and Coffman, E. G., "A Study of Interleaved Memory", AFIPS Proc. SJCC, Vol. 36, pp. 467-474, 1970.

(35) Burnett, G. J., and Coffman, E. G., "Analysis of Interleaved Memory Systems Using Blockage Buffers", Commun. ACM, Vol. 18, No. 2, pp. 91-95, February 1975.

(36) Skinner, C., and Asher, J., "Effect of Storage Contention on Performance", IBM Syst. J., Vol. 8, No. 4, pp. 319-333, 1969.

(37) Ravi, C. V., "On the Bandwidth and Interference in Multiprocessors", IEEE Trans. Comput., Vol. C-21, pp. 899-901, August 1972.

(38) Bhandarkar, D. P., "Analysis of Memory Interference in Multiprocessors", IEEE Trans. Comput., Vol. C-24, pp.897-908, September 1975.

(39) Sastry, K. V., and Kain, R. Y., "On the Performance of Certain Multiprocessor Computer Organizations", IEEE Trans. Comput., Vol. C-24, pp. 1066-1074, Nov. 1975.

(40) Baskett, F., and Smith, A., "Interference in Multiprocessor Computer Systems with Interleaved Memory", Commun. ACM., Vol. 19, No. 6, pp.327-334, June 1976.

(41) Briggs, F. A., and Davidson, E. S., "Organization of Semiconductor Memories for Parallel - Pipelined Processors", IEEE Trans. Comput., Vol. C-26, pp. 162-169, February 1977.

(42) Briggs, F. A., "Memory Organizations and Their Effectiveness for Multiprocessing Computers", Coordinated Science Lab., Report No. R-768, Univ. of Ill., May 1977.

(43) Bloom, L., Cohen, M., and Porter, S., "Considerations in the Design of a Computer with High Logic - to - Memory Speed Ratio", Proc. Gigacycle Computing Systems, January 1962; AIEE Special Publ., S-136, pp. 53-63.

(44) Liptay, J. S., "Structural Aspects of the System/360 Model 85, Part II: The Cache", IBM Syst. J., Vol. 7, pp. 15-21, 1968.

(45) Strecker, W. D., "Cache Memories for PDP-11 Family Computers", The 3rd Annual Symposium on Computer Architecture, pp. 155-158, January 1976.

(46) Kaplan, K. R., and Winder, R. O., "Cache - Based Computer Systems", Computer, pp. 30-36, March 1973.

(47) Meade, R, M., "On Memory System Design", AFIPS Proc. FJCC, Vol. 37, pp. 33-43,1970.

(48) Sisson, S. S., and Flynn, M. J., " Addressing Patterns and Memory Handling Algorithms", AFIPS Proc. FJCC, Vol. 33, Part 2, pp. 957-967. 1968.

(49) Smith, A. J., "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory", IEEE Trans. Software Eng., Vol. SE-4, No. 2, pp. 121-130, March 1978.

(50) Rao, G. S., "Performance Analysis of Cache Memories", J. ACM, Vol. 25, No. 3, pp. 378-395, July 1978.

(51) Gibson, D. H., "Considerations in Block - Oriented Systems Design", AFIPS Proc. SJCC, Vol. 30, pp. 75-80, 1967.

(52) Conti, C. J., "Concepts for Buffer Storage", IEEE Comput. Group News, Vol. 2, pp. 9-13, March 1969.

(53) Belady, L. A., "A Study of Replacement Algorithms for Virtual Storage Computer", IBM Syst. J., Vol. 5, pp. 78-101, 1966.

(54) Belady, L. A., Nelson, R. A., and Shedler, G. S., "An anomaly in the Space - Time Characteristics of Certain Programs Running in Paging Machines", Commun. ACM, Vol. 12, No. 6, pp. 349-353, June 1969.

(55) Bell, J., et al., " An Investigation of Alternative Cache Organizations", IEEE Trans. Comput., Vol. C-23, No. 4, pp. 346-351, April 1974.

(56) Smith, A. J., "Characterizing the Storage Process and Its Effect on the Update of Main Memory by Write Through", Commun. ACM, Vol. 26, No. 1, pp. 6-27, January 1979.

(57) Pohm, A. V., et al., " The Cost and Performance Tradeoffs of Buffered Memories", Proc. of the IEEE, Vol. 63, No. 8, pp. 1129-1135, August 1975.

(58) Erhan Ginlar, Introduction to Stochastic Processes, Prentice - Hall, Inc., 1975.

(59) Chang, D., et al., "On the Effective Bandwidth of Parallel Memories", IEEE Trans. Comput., Vol. C-26, No. 5, pp. 480-490, May 1977.

(60) Emer, J. S., "Shared Resources for Multiple Instruction Stream Pipelined Processors", Coordinated Science Lab., Report No. R-838, Univ. of Ill., July 1979.

(61) Birtwistle, G. M., et al., SIMULA Begin, Van Nostrand Reinhold, Second Edition, 1979.

(62) Easton, M. C., and Fagin, R., "Cold-Start vs. Warm-Start Miss Ratios", Commun. ACM, Vol. 21, No. 10, pp. 866-872, October 1978.

(63) Patel, J. H., Private Communication.

(64) Chow, C. K., "On Optimization of Storage Hierarchies", IBM J. Res. Dev., pp. 194-203, May 1974.

(65) Welch, T. A., "Memory Hierarchy Configuration Analysis", IEEE Trans. Comput., Vol. C-27, No. 5, pp. 408-417, May 1978.

VITA

Chi-Chung Yeh was born in Taiwan, Republic of China, on June 12, 1950. He received a B. Eng. degree in Electronic Engineering from Chung Yuan Christian College for Science and Engineering, Taiwan, Republic of China, in 1972.

He received an M.S. degree in Electrical Engineering from Northwestern University, Evanston, Illinois, in 1975 and an M.S. degree in Computer Science from the University of Illinois at Urbana-Champaign, in 1977.

From 1978 to 1980, he was a graduate assistant at the Coordinated Science Laboratory of the University of Illinois at Urbana-Champaign.


DATE
ILME