INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9130361

Performance analysis of the memory hierarchy via Markov chains and state cloning

Peterson, Edward F., Ph.D.

City University of New York, 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A

PERFORMANCE ANALYSIS OF THE MEMORY HIERARCHY VIA MARKOV CHAINS AND STATE CLONING

by

Edward F. Peterson

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1991

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

5/10/91
Date

Jacob Rootenberg TW
Dr. Jacob Rootenberg
Chair of Examining Committee

5/10/91
Date

T C Wesselkamper
Dr. T. C. Wesselkamper
Executive Officer

Dr. Michael Anshel

Dr. Cyrus Mohebbi

Dr. Jerry Waxman

Dr. T. C. Wesselkamper

Supervisory Committee

The City University of New York

ABSTRACT

PERFORMANCE ANALYSIS OF THE MEMORY HIERARCHY VIA MARKOV CHAINS AND STATE CLONING

by

Edward F. Peterson

Adviser: Professor Jacob Rootenberg

The performance of the memory hierarchy currently plays an important role in the design of computer systems. Most computers today, from micros to mainframes, utilize cache memory to increase overall system performance. Parallel processing systems sometimes use cache memory to allow all processors to share the same address space. Even super computer design is influenced by the performance of the memory hierarchy.

This dissertation presents an innovative approach to the performance evaluation of hierarchical memory systems. Traditional evaluation techniques of such systems employ simulation. Simulation is often expensive because it is extremely time consuming and requires much disk space. An alternative approach, through the use of Markov Chains and state cloning techniques, is presented in this dissertation.

The memory hierarchy is explained and a performance model is developed. This performance model is applied to traditional cache memory. Shared memory in parallel processing systems is presented along with the concerns of memory coherency. Memory coherency protocols are given and the newly developed performance model is used to measure and to compare the efficiency of the given protocols.

Dedication

This dissertation is dedicated to the memory of Professor Jacob Rootenberg whose valuable guidance and constant encouragement kept me inspired throughout this research. Dr. Rootenberg was both an advisor and a friend who was able to accomplish the near impossible task of providing superb technical guidance in an enjoyable manner. Dr. Rootenberg engaged me in many intellectually challenging conversations regardless of place or time, whether in a restaurant in Sheepshead Bay Brooklyn, strolling along the ocean on Manhattan Beach, or more commonly meeting late at night in his office. I am grateful to have had the pleasure of working with him.

Table Of Contents

List Of Tables

List Of Figures

Chapter 1

Introduction

1.1 Overview

Recently there have been several advances in the design of low cost high speed computers. These advances have allowed micro computers to achieve performance levels equal to, if not greater than that of large main frame computers [Simm90]. Even the most current super computer technology is centered around a collection of low cost micro computers [Seit85] and [Seit89]. One of the most rapid growing areas of new processor technology is the very fast execution of a small set of instructions. Most noteworthy is the Reduced Instruction Set Computer (RISC) [Patt85]. RISC computers have a reduced (reasonably simple) set of instructions and thus can operate at faster speeds than their more traditional counterparts, the CISC (Complex Instruction Set Computer) computers. Due to their reduced complexity, RISC computers can also be designed in a much shorter time than CISC computers. For these reasons the most common design method used today for micro and mini computers is that of the RISC architecture. The current emphasis is on ultra-fast instruction execution, even if it means that the instructions perform a much less complicated task. Groups of smaller, simpler instructions can be executed to do larger

and more complex instructions (tasks). With higher speed instruction execution, memory access must also be comparably fast. Small high-speed memories are used to keep up with the fast processor. These high-speed memories, commonly known as cache memories, are used as buffers to bridge the large speed difference between the processor and main memory.

It is common for today's microprocessors, such as the Intel i860 [Inte90], to have on-chip instruction and data cache memories to facilitate the required CPU's execution speed. Some computers even have multiple levels (hierarchy) of cache memory to optimize both memory speed and size at an economical acceptable cost. This hierarchy would allow for both a very fast and very large memory address space at an acceptable cost. As these microcomputer systems are grouped together to become larger parallel processing systems, cache design, performance, and coherency become major issues. A memory is said to be coherent when a read operation returns the value written from the most recent write operation. Memory coherency protocols and algorithms must be used in the implementation of shared-memory for loosely-coupled parallel processing systems. These algorithms play an important role in the overall performance of the system.

This dissertation develops an analytical model that can be used for analyzing the performance of the memory hierarchy. The model developed in this thesis is based on Markov chains and state cloning techniques, specifically developed for modeling the various alternatives of memory hierarchies. A model is first introduced through an example of a traditional memory hierarchy which contains three levels of memory. Some design improvements, i.e., a better page replacement algorithm or a larger cache, that would affect the performance of the memory system are then applied to the memory hierarchy. The model is subsequently enhanced to allow for non-equal fixed time delays, which allow the model to include the additional complexities of the new and improved memory

hierarchy. In a non-equal fixed time delay, each level in the memory hierarchy can contribute a different amount of time to the total memory access time. The new memory system is modeled and the results are compared with the performance of the prior system. The memory hierarchy is then further enhanced, this time to a more realistic situation of variable time delays within a given level of memory. The access-time associated with cache memory access is entirely different, depending upon whether or not the desired memory location currently resides in the cache. The model used to evaluate the performance is then enhanced again by developing a new state cloning technique which models variable time delays. The performance of the new hierarchical memory example is computed and the results are compared with the performance of the prior system. The performance model is now at the complexity level needed to handle all forms of hierarchical memory systems, including that of shared-virtual-memory.

Shared-virtual-memory [Li86b] is a form of memory used with loosely-coupled processing systems, i.e., systems that cannot directly access a common memory location, that transforms all or part of their private independent memories into a global memory that all can access. Global shared memory gives software developers alternate avenues for developing applications by providing access to memory, regardless of its physical location, i.e., a globally addressed memory shared by all processors. It also allows the system to handle in a general way, complex low level memory management tasks. The existence of a shared-virtual-memory on a loosely-coupled processing systems implies that a single page of memory can also coexist on multiple processors simultaneously, and memory coherency protocols must therefore be used to ensure data consistency. The most common protocol used on these type systems is the write-back page-invalidation protocol, which allows many read copies of a page to exist but only one write copy. This is similar to the readers and writer problem where many readers or one writer can access memory. Obviously memory coherency can be maintained by allowing the existence of

only one, fixed or floating, physical location of any given logical memory address. This would not be an efficient algorithm for many reasons, such as the common use of static read-only data, in general purpose applications.

An efficient shared-virtual-memory algorithm would allow multiple copies of a page of memory to coexist simultaneously. With multiple copies of a memory location, each write operation must write to all of the existing copies in order to maintain memory coherency. The task of ensuring memory coherency could be extremely time consuming and thus become grossly inefficient. For the reasons described above, a write-back with page-invalidation approach is the most efficient algorithm for general purpose applications. In this last approach, when a memory location logical address is requested for writing, a copy of the page which contains the memory location (physical address) is given to the requesting processor and all other copies of that page are invalidated. This ensures that only one copy of the page exists when a write operation is performed. Write-back refers to the algorithm allowing multiple writes to the given local page and, at some later point in time, when another processor requests it, the page is copied to the processor's attached local memory, as opposed to a write-through algorithm which allows for the updates of other copies of a given page, whenever a write operation takes place.

In a write-back page invalidation of a shared-virtual-memory algorithm, every page must have an owner. The function of the owner is to maintain a copy of the page. When other processors request read access to a page, the owner must provide a copy of the page. When a processor requests write access to a memory location, that processor is given a copy of the page which contains the requested memory location, all other copies of the page are invalidated, and the requesting processor is made the new owner of the page. That processor must become the owner because at that point, during a write operation, no other valid copies of the page exist. Invalidating all other copies of the page

implies that some processor must keep track of where all copies of a given page reside. This management task of maintaining a list of which processors currently have access to a given page is the responsibility of the page manager. The page management function can either be centralized or distributed. Distributed page managers can be assigned pages to manage on a static or dynamic basis. The operation of the page manager that is used is a main factor in the performance of a memory coherency protocol.

The hierarchical memory performance model is used to model the performance for a read operation on a shared-virtual-memory. The read operation is less complex than a write operation since a read never causes a page invalidation to take place. The performance of a write operation on a shared-virtual-memory is then analyzed and compared to that of the read. Next, the performance of a write operation on a shared-virtual-memory with a variable time invalidation function is analyzed and the results are compared with the previous system. The memory coherency protocol used in the previously modeled system is then changed to contain a distributed page manager, the system is modeled and the results are compared. An enhancement is then made to the memory coherency protocol which dramatically increases the performance of the shared-virtual-memory system. The new system is modeled, the results are compared, and it is proven to be a good enhancement.

The research outlined in this thesis develops an analytical framework for analyzing the performance of the memory hierarchies. The hierarchical memory performance evaluation framework, developed here, could be used by system designers to quantify the impact of hierarchical memory design options before actual implementation. This framework could also be used when designing memory coherence protocols. The next section gives a detailed description of the organization and contents of this dissertation.

1.2 Thesis Organization

Chapter 2 gives a description of the memory hierarchy. This description includes cache memory, main memory, and page replacement strategies in cache memory. Chapter 2 also includes an introduction to memory coherency along with what coherency means for message passing multiprocessors.

Chapter 3 presents an innovative approach to the performance evaluation of hierarchical memory systems. The performance model developed utilizes Markov chains and state cloning techniques to model variable time delays. The notion of system availability is defined and used in analyzing several hypothetical systems. The performance model is first presented with an example of a simple memory system. The example memory systems and model are then grown in complexity until the full performance model is presented and complex memory systems are evaluated.

Chapter 4 introduces the concept of shared-virtual-memory, along with a discussion of why it is needed and how it is implemented. The meaning of memory coherency in a shared-virtual-memory setting is defined. A simple algorithm for a shared-virtual-memory system is presented. A more efficient algorithm is also given, examined and analyzed.

Chapter 5 uses the performance model, from chapter 3, for analyzing the performance of the shared-virtual-memory system described in chapter 4. The most simplistic part of the shared-virtual-memory system, a read instruction, is analyzed first. Next the write instruction of the most basic shared-virtual-memory algorithm is analyzed. The write instruction takes into account invalidation where the read instruction does not. The prior write operation on a shared-virtual-memory is then enhanced to include the more realistic case of a variable time page invalidation function. The results of the two write systems, from a write-availability point of view, are then compared. A distributed page

manager function is then incorporated, followed by a more efficient memory coherency protocol which is analyzed as well.

Chapter 6 summarizes the dissertation, presents related work in this area, and gives possible directions for future research in this area.

Chapter 2

The Memory Hierarchy

Today CPU instruction execution speed is faster than ever before. New microprocessors, utilizing super scalar architecture and executing multiple instructions per cycle are currently available [Kohn89] and [Grov90]. It is projected that by the year 2000, microprocessors will be able to execute 2,000 million instructions per second (MIPS) [Gels89]. Due to technology/cost choices, main memory access time is considerably slower than CPU cycle time.

2.1 Cache Memory

Cache memories are used as small high-speed buffers, located logically between the CPU and main memory to bridge the gap between the fast cycle time of the CPU and the relatively slower access-time of main memory. When the CPU requests the contents of a memory location, the cache services that request immediately if the desired location currently exists in the cache. If the desired location is not in the cache, the appropriate block of memory, which contains the desired location, is loaded from main memory into the cache and then the memory request is serviced. Cache memory thus provides fast access when the desired memory location resides in the cache. Cache memory should

also provide a high hit-ratio (i.e., percentage of requests for memory locations that reside in the cache verses total requests issued) to reduce the impact of slow main memory. Large caches with high hit-ratios are slower and have a higher access-time for memory locations that reside in the cache than smaller caches with lower hit-ratios. Multiple levels of cache are used to obtain both a high hit-ratio and fast memory access at a reasonable economic price. A memory structure with multiple levels of cache is shown in Figure 2.1. Each level in the memory hierarchy may be invoked depending upon whether or not the previous level of memory contained the information desired. If such memory access ended up in success (hit), i.e., the memory location exists in the current level of the memory hierarchy, no more levels of memory are invoked. If, however, the desired location is not found in the current level of memory (miss), then the next level of the memory hierarchy is invoked. This process continues until the required location is found. Sometimes the desired location resides in the memory level closest to the CPU, the fastest cache, while other times multiple levels of the memory hierarchy are needed to be accessed in order to fetch the desired data. When the desired memory location is eventually found it must be transferred up to the next level of memory closer to the CPU. This process continues until the content of the memory location sought by the CPU, exists in the memory level adjacent to the CPU (it is from this level that CPU access takes place). For more detailed information on multilevel cache hierarchies refer to [Baer89], [Bril87], and [Wan89b].

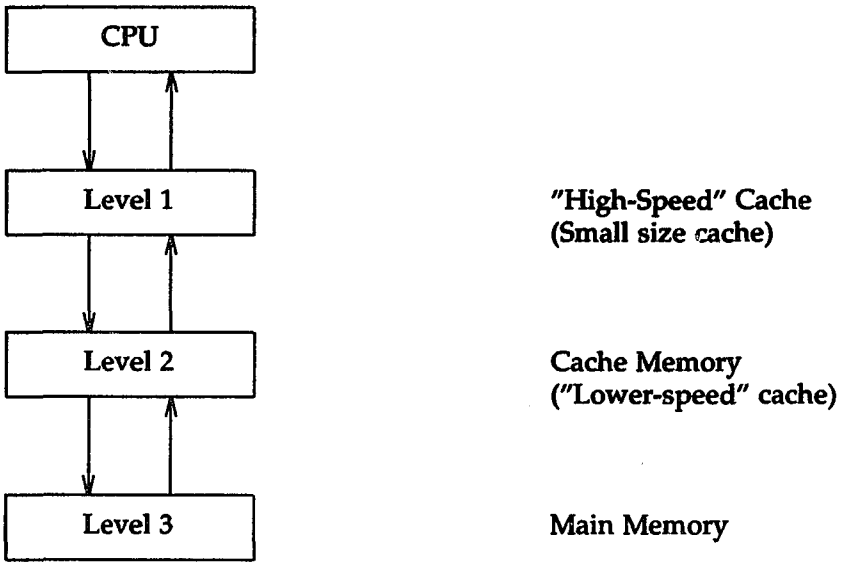


Figure 2.1: A model of memory hierarchy.

2.2 Page Replacement Strategies

When transferring data into cache memory, a page which currently resides in the cache must be selected for replacement. This replaced page makes room for the new page of memory which contains the desired location. The algorithm used to select a page for replacement is referred to as the page replacement strategy. The most common strategy used is the Least Recently Used (LRU) page replacement algorithm. This algorithm works well when computer programs frequently access the same memory location or memory locations close to ones previously accessed. Accessing information which is close to information previously accessed is referred to as the "principle of locality". Other replacement strategies such as First In First Out (FIFO) or Most Frequently Used (MFU) algorithms are sometimes used as well. A page chosen for replacement might require a memory transfer or write-back to main memory. This is the case if the replacement page has been modified and the modifications have not been written back to main memory.

Some caches implement a "write-through" scheme which writes modifications back to main memory, or to the next level of cache closer to main memory. This write-through occurs every time a modification to the data is made. When a cache uses a write-through strategy of memory updating, replacement pages need not be written back to main memory. The information in the cache is already consistent with main memory at the time of replacement. Sometimes a cache is read-only, in which case writing information back is never required. One very common usage of Read-only caches is to store instructions for the computer. These read-only caches are commonly found on-board microprocessors. The Intel i860 [Inte90] and the IBM RS6000 [Hard90] have two built in caches, one read-only and the other read-write.

2.3 The Cache Coherency Problem

When cache memory is used in conjunction with multiprocessing systems, it is typically configured such that each processor has a private cache [Min89] and [Squi90]. If these caches are read-write, extra provisions must be made to ensure cache coherency. A cache is said to be coherent if a read from the cache always returns the information most recently written to that location. In a multiprocessor system with shared memory, i.e., multiple processors can access the same memory address, the same memory location can concurrently exist in multiple sites, i.e., multiple caches. If one processor updated a memory location whose data also existed in the cache of the other processor(s), the other processor(s) cache must also be updated. This is to ensure that information returned from a cache always is the same information that was most recently written to that particular memory location. The process used in keeping all cache memory consistent in a multiprocessor, multi-cache architecture is the cache coherency protocol. Some multiprocessor configurations connect all caches to memory via the same bus as shown in figure 2.2.

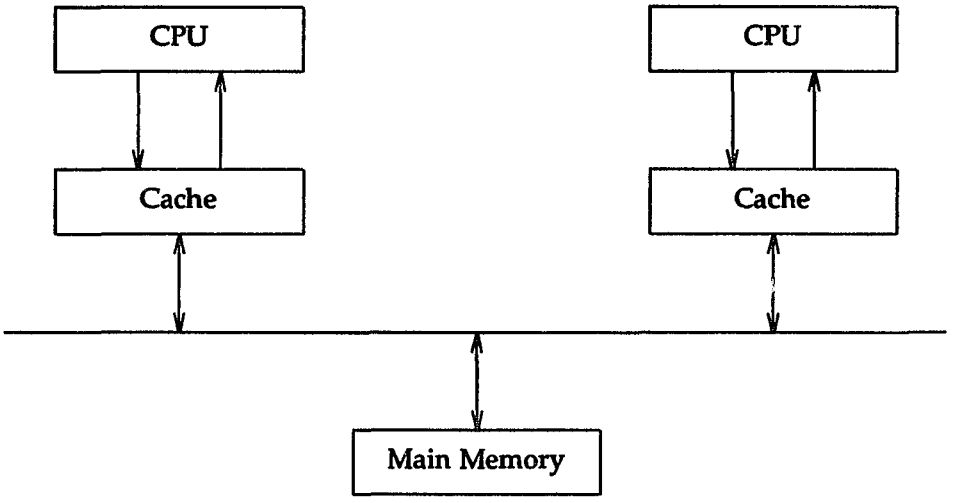


Figure 2.2: A Multi-CPU Multi-Cache Architecture.

An example of a cache coherency protocol for the configuration shown in figure 2.2 could be to have all caches implemented as write-through caches which also listen to the traffic on the bus. If a cache hears a memory update on the bus which is for a memory location that currently resides in that cache, it updates it's location with the information heard on the bus. This is sometimes referred to as a "Snoopy Cache" for the way in which the cache snoops on the bus. For more information on the "Snoopy Cache" and other multiprocessor cache coherency algorithms, the reader may refer to [Sche87], [Swea85], or [Wils87].

2.4 Coherency For Message Passing Multiprocessors

Message-passing multiprocessing systems such as the Intel iPSC/2 hypercube [Arla88] need to address the issue of memory coherency differently. Each processor in a message-passing system are self contained computers with private memory as shown in figure 2.3.

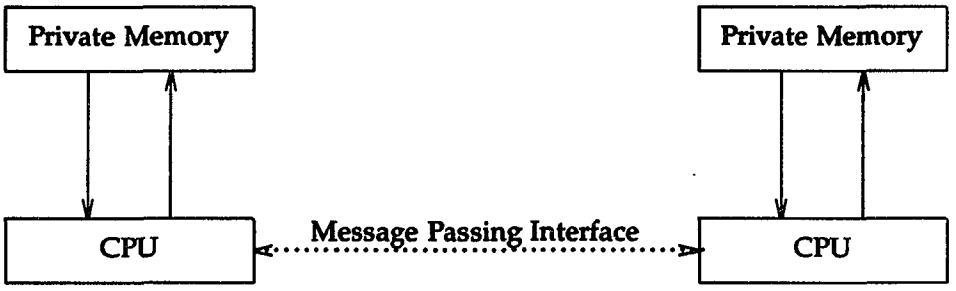


Figure 2.3: Message Passing Processors

Shared memory in a message passing system would be implemented virtually through software [Li86a]. Keeping the memory coherent would involve either a memory page invalidation or write-back approach. In the invalidation approach, a write to a memory location first invalidates all other pages which contained that location. After invalidation is complete the actual write occurs to the one remaining page, i.e., all other pages are invalid. After the invalidation has occurred, a reference to that memory location from a processor which had it's page invalidated requires a new copy of the page to be moved to that processor's cache. In the write-back approach, a memory-write writes to all sites that have a copy of the page, updating each copy of the affected page. The time delay associated with processor to processor communication usually makes the write-back approach too costly. The topic of cache coherency for message passing multiprocessors is examined thoroughly in chapter 5 of this dissertation.

Chapter 3

The Memory Performance Model

This chapter presents an innovative approach to the performance evaluation of hierarchical memory systems. Traditional evaluation techniques of such systems employ simulation. Simulation is often expensive because it is extremely time consuming and requires much disk space. The evaluation of such systems is a difficult task without the use of simulation techniques.

An alternative approach through the use of Markov Chains and state cloning techniques is presented here. The approach is explained and is used in analyzing the performance of hierarchical memory systems. This approach could help system designers quantify the cost associated with a particular design while it is still under consideration. It could also be used to quantify the impact associated with design options, so that competing alternatives can be compared and a optimal design thus may emerge.

3.1 Overview

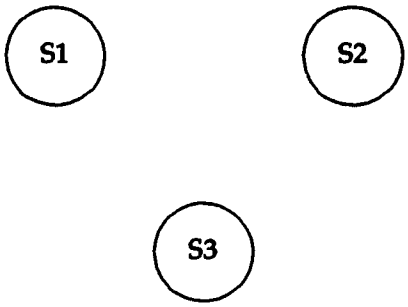
When designing or implementing a hierarchical memory system it is always important and sometimes vital to quantify the performance of design alternatives. This way,

the system could be optimized and the appropriate design decisions could be made. Having a design tool to compare plausible alternatives will no doubt help in evaluating the appropriate design decisions that can be adopted and lead to an optimal implementation. Traditional evaluation techniques of such systems employ trace-driven simulation [Wang88]. Trace-driven simulation is often expensive because it is an extremely time consuming process that requires massive amounts of disk storage [Wan89a]. This chapter presents an alternative method for evaluating the performance of hierarchical memory systems.

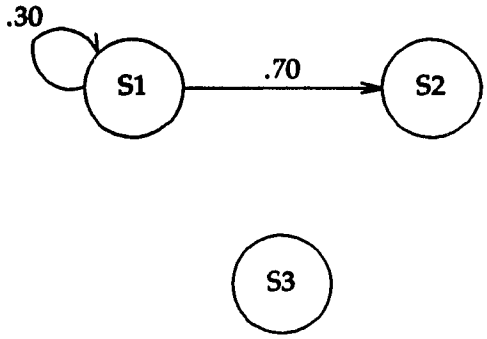
3.2 The Performance Model

In this section, we examine a three level model of memory hierarchy, two levels of cache and one level of main memory. We assume, for the time being, that each level of memory utilized contributes one cycle of processing time to the total time required for memory access. In addition, for our example to be concrete, we further assume some particular statistics such as for instance that 30% of memory access requests eventually utilize one level of memory (i.e., the highest level of cache), 45% of all memory access requests end up utilizing two levels of memory, and the remaining 25% of memory access requests need all three levels of memory. We now construct a state-transition diagram which describes this model. Figure 3.1a gives the mapping of hierarchical memory levels to transition states. State S1 represents the level of memory which is closest to the CPU This memory level is usually implemented as a very fast cache. State S2 represents the next memory level, i.e., the higher capacity slower speed cache, which is searched only when the desired memory location is not resident in the prior level. The last level of memory, state S3, contains every memory location and is utilized only when a desired memory location is not present in prior levels. Figure 3.1b shows all possible transitions

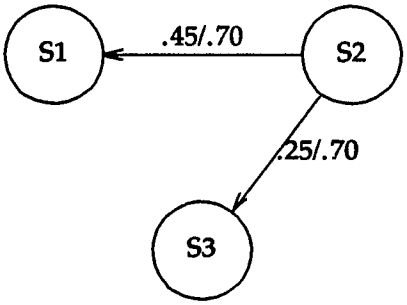
from state S1. The arc from S1 to itself represents 30% of all memory requests which are serviced in the highest (closest to the CPU) level of memory and thus if a hit is found, the next memory access attempts to again access this level. The arc from S1 to S2 represents the remaining 70% (i.e., 45% two-level access and 25% three-level access) of memory requests that mandate additional levels of memory. Figure 3.1c shows the possible transitions from state S2. The arc from S2 to S1 represents memory access which requires exactly two levels of memory. This occurs in 45% of all memory access. The arc is given a probability of .45/.70, which is the probability of level 2 memory requests that only require level 2. This is equivalent to 45% of all memory requests. The arc from S2 to S3 represents the 25% of the total memory requests which require all three levels of memory and is labeled with the probability of .25/.70 . It should be noted that a normalization must be performed (division by .70) so that the sum of the probabilities of all arcs leaving a given node equals one. Figure 3.1d illustrates the fact that all memory requests that eventually require memory level 3 are satisfied in that level, no miss condition can occur in level 3 since level 3 contains all entries. The arc from state S3 to state S1 has a probability of 1.0 which indicates the fact that no miss can occur in this level. The memory access is definitely satisfied once this level is reached because, as stated before, all memory locations reside in this level. The path from S1 to S2 to S3 to S1 contains three arcs which represents the three cycles or time slices which are required to satisfy a level 3 memory request. This is consistent with our initial assumption that each level of memory adds one cycle of time and a three level memory request takes three cycles. The actual path that the memory location would take is to be copied from level 3 to level 2 and then from level 2 to level 1 where the actual memory reference by the CPU occurs. Figure 3.1e gives the complete state-transition diagram. It should be noted that state S1 represents the initial state which all memory requests must go through.



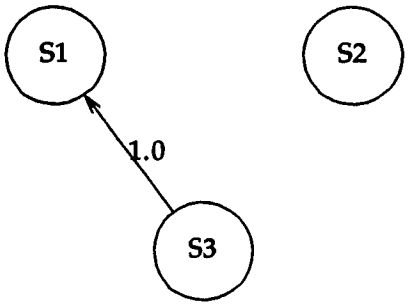
States of the model
Figure 3.1a



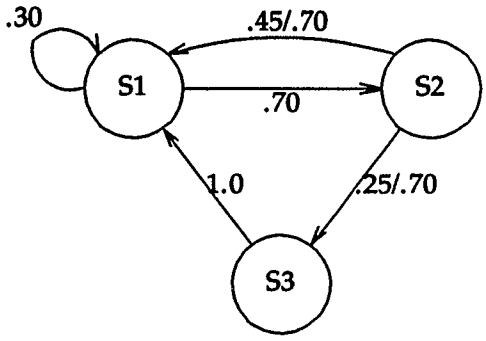
Transitions from state S1
Figure 3.1b



Transitions from state S2
Figure 3.1c



Transitions from state S3
Figure 3.1d



State-transition diagram for a three-level memory hierarchy
Figure 3.1e

The arcs in figure 3.1e are now explained:

S1 -> S1: The arc from S1 to S1, labeled .30, represents the 30% of all memory access requests that utilize only one level of memory. The desired memory location exists in the highest level of memory, i.e., the memory level adjacent to the CPU.

S1 -> S2: The arc from S1 to S2, labeled .70, represents the 70% of all memory access requests that requires more than one level of memory. The desired memory location does not exist in the highest level of memory, and at least one, if not more, additional levels of memory are needed.

S2 -> S1: The arc from S2 to S1, labeled .45/.70, represents the 45% of all memory access requests that requires exactly two levels of memory. The desired memory location does not exist in the highest level of memory but does in the second level of memory.

S2 -> S3: The arc from S2 to S3, labeled .25/.70, represents the 25% of all memory access requests that requires more than two levels of memory. The desired memory location does not exist in the first two levels of memory.

S3 -> S1: The arc from S3 to S1, labeled 1.0, represents the 100% of all memory level three requests that does not require any more levels of memory, i.e., all memory requests that are not satisfied in memory level one or memory level two, must be satisfied in memory level three.

After constructing the state-transition diagram (see figure 3.1e) our next step in analyzing the hierarchical memory system is to build a state-transition matrix. This is a square matrix with n rows and n columns where n is the number of states in the system. Each element in the matrix is the probability of making a transition from the state associated with the row (labeled as: "from") to the state associated with the column (labeled as:

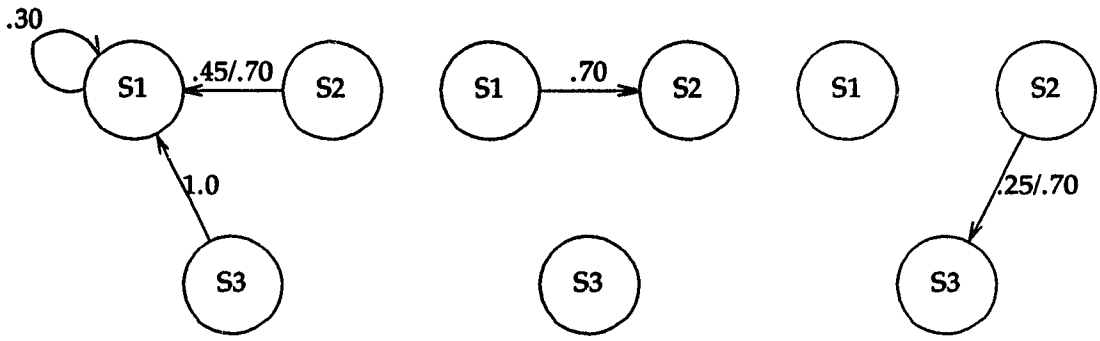
"to"). The sum of the probabilities in each row of the transition matrix must naturally be equal to one, since the sum of all transition probabilities from any given state, to all other states reachable from that given state, must be equal to one. Note that the above does not preclude a state coming back to itself like state S1 in figure 3.1e. The state-transition matrix for our three-level memory hierarchy is given in Table 3.1.

FROM	TO		
	S1	S2	S3
S1	0.3	0.7	0.0
S2	.45/.70	0.0	.25/.70
S3	1.0	0.0	0.0

Table 3.1: State-transition matrix for the model of figure 3.1e.

3.3 System Availability

To determine the cache memory system's performance we need to compute the availability of the system and its average processing time. Being in state S1 implies that the hierarchical memory system is available and ready to accept a memory request. This request may require one, two, or all three levels of memory access. Determination of system availability is equivalent to determining the probability of the memory system in the model depicted earlier being in state S1. Examining the structure in figure 3.1e, the arcs emanating from all states to a given state are indicative of the probability of being in the state that the arcs lead into. Thus, for instance a composite sequence corresponding to figure 3.1e can be written and expressed as in figure 3.2.



S1 Probability Figure 3.2a **S2 Probability Figure 3.2b** **S3 Probability Figure 3.2c**

From this description, it should thus be clear (see figure 3.2a) that the probability of the memory access system being in state S1 (P(S1)) is equal to .30 times the probability of the system being in state S1 plus (.45/.70) times the probability of the system being in state S2 plus 1.0 times the probability of the system being in state S3. This is so, because when the system is in state S1, 30% of all memory references causes the system to make a transition back to state S1 (i.e., the memory request is serviced in the highest level of cache); (45/70)% of memory references when the system is in state S2 brings the system back to state S1 and all memory references (100%) in state S3 is serviced there without the need of additional levels of memory. We can now compute the probabilities of the system being in states S2 and S3 in exactly the same manner following the structures exhibited in figure 3.2b and 3.2c. Each of the state probabilities is given below.

$\hat{P}(S1)$ = .3 P(S1) + (.45/.7) P(S2) + P(S3)
$\hat{P}(S2)$ = .7 P(S1)
$\hat{P}(S3)$ = (.25/.7) P(S2)

P(Si) represents the probability that the system is in state Si and $\hat{P}(Si)$ represents the probability that in the "next state" the system will be in is the state Si. The "next-state" probabilities, given above, describe the probability of being in the next state, assuming we know the probability of being in the current state. They depict a "one transition"

probability. We now solve for the current-state probability of being in state S1 by using the fact that the next state of the system must be one of the three states of the model. The sum of the "next-state" probabilities of all three states must thus equal one.

$$\hat{P}(S1) + \hat{P}(S2) + \hat{P}(S3) = .3\ P(S1) + (.45/.7)\ P(S2) + P(S3)$$
$$+ .7\ P(S1)$$
$$+ (.25/.7)\ P(S2)$$

$$1.0 = .3\ P(S1) + (.45/.7)\ (.7)\ P(S1)$$
$$+ (.25/.7)\ (.7)\ P(S1)$$
$$+ .7\ P(S1)$$
$$+ (.25/.7)\ (.7)\ P(S1)$$

$$1.0 = 1.95\ P(S1)$$

$$P(S1) = .51282$$

The solution P(S1) indicates that for this choice of probability parameters the probability of the system being in state S1 and thus available to immediately process requests is .51282. The longest wait is 3 cycles and the average wait is 1.95 cycles. This availability, of the memory hierarchy, can also be solved through state-transition matrix multiplication as shown below. Let P(i,j) be the probability of making a transition from state i to state j where i is the state associated with the row and j is the state associated with the column, then:

$$P(i,j) = \begin{vmatrix} 0.30000 & 0.70000 & 0.00000 \\ 0.64286 & 0.00000 & 0.35714 \\ 1.00000 & 0.00000 & 0.00000 \end{vmatrix}$$

$P(i,j)$ is thus the probability of making a transition from state i to state j in the first cycle. The probability of making a transition from state i to state j after two cycles is: $P(i,j)^2$ and it is equal to the sum of $P(i,k) * P(k,j)$ for all states k.

$$P(i,j)^2 = \sum_k P(i,k) * P(k,j) \ .$$

Solving this equation for all i and j is equivalent to the multiplication of the transition matrix by itself. The probability of making a transition from state i to state j in (m+n) transitions is given by the Chapman-Kolmogorov equation.

$$P(i,j)^{(m+n)} = \sum_k P(i,k)^m * P(k,j)^n \quad .$$

For a complete derivation of the Chapman-Kolmogorov equation refer to [Fell68] or [Klei75]. In general, the probability of making a transition from state i to state j in n cycles can be computed by raising the transition matrix to the power of n. The probability of making a transition from state i to state j after 2 cycles is given below as:

$$P(i,j)^2 = \begin{vmatrix} 0.54000 & 0.21000 & 0.25000 \\ 0.55000 & 0.45000 & 0.00000 \\ 0.30000 & 0.70000 & 0.00000 \end{vmatrix}$$

The system stabilizes to within 5 significant decimal places after 22 cycles, i.e., the difference between $P(i,j)^{22}$ and $P(i,j)^{23}$ is less than 0.000005 for all i and j. Systems which stabilize in this manner are known as regular [Varg62]. When the system stabilizes, a row of this matrix will form a left eigenvector of the initial transition matrix with eigenvalue 1 [Sene73]. This can be used as a check to ensure that the correct answer is computed.

$$\hat{P}(i,j)^{22} = \begin{vmatrix} 0.51282 & 0.35897 & 0.12821 \\ 0.51282 & 0.35897 & 0.12821 \\ 0.51282 & 0.35897 & 0.12821 \end{vmatrix}$$

These are the "next-state" probabilities of the system which remain (almost) constant for any further number of cycles, or powers of the transition matrix, i.e., cycles that are greater than 22. When the next-state probabilities become constant, they are referred to as the steady-state probabilities. If the steady-state probabilities of making a transition to a given state j are the same from all states i, then that probability are the current-state

probability of state j. Note that we arrive at the same current-state probability for state S1 as when we solved for the current-state probability using the next-state probability equations.

3.4 Modeling Non-Equal Fixed Delays

In the prior example we examined a multilevel memory hierarchy where each level of the memory hierarchy had an equal processing time or delay of one cycle. We now analyze a much more complicated situation, a system with non-equal processing time, i.e., the delay contributed by each level of the memory hierarchy could be different. For the purpose of comparison, we modify the example given in the previous section and add an additional delay to two of the memory levels. Our example, as before, consists of three levels of memory modules which are labeled as L1, L2, and L3, corresponding to states: S1, S2, and S3 respectively. If we assume that levels L2 and L3 have a processing time of two cycles each and L1 still has a processing times of 1 cycle, then our new system processes 30% of memory access requests in one cycle, 45% of all memory access requests in three cycles and 25% of all memory requests in five cycles. Note, when we utilize level L3 of the memory system we must also utilize level L2 and Level L1. In our memory system, level L1 is used by all requests, level L2 is used by the remaining 70% (45% + 25%) of all memory requests and level L3 is only used by 25% of all memory requests. In order to solve for the system availability of the new memory system we must first clone states S2 and S3 in order to represent the additional delay unit needed in each of these states.

Figure 3.3a shows state S2, before cloning, with all state S2 entry and exit arcs. In figure 3.3b state S2 is split into two states, S2a and S2b, connected by an arc, labeled 1.0. The arc connecting states S2a and S2b represents the additional time unit incurred by all

memory references that utilize level L2 of memory. Since a probability of 1.0 is assigned to the S2a-S2b arc, all transitions that enter S2a must next enter S2b. For this reason, all arcs entering the initial state S2, in figure 3.3a, must now enter state S2a as shown in figure 3.3c. Also for the same reason stated above, all arcs initially exiting from the original state S2 must now exit from the new state S2b. The complete cloning of state S2 into states S2a and S2b is given in figure 3.3d.

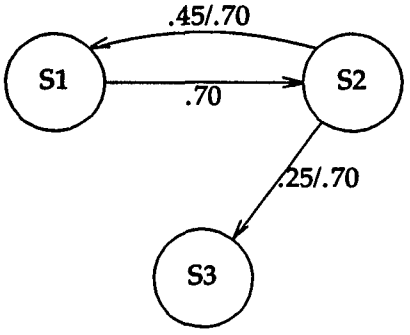


Figure 3.3a
State S2 Before Fixed Clone

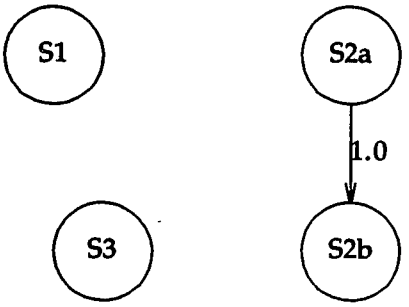


Figure 3.3b
State S2 After State Split

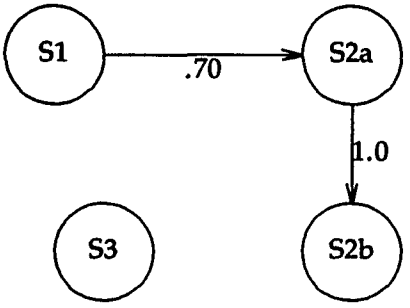


Figure 3.3c
State S2 With Proper Input

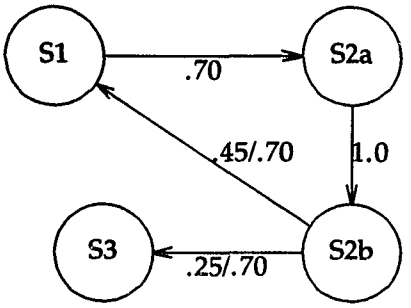


Figure 3.3d
State S2 After Fixed Delay Clone

State S3 is cloned into S3a and S3b as show in figure 3.4. States S2 and S3 were cloned since they are the states that underwent the change from the old one cycle

processing time to the current two cycles of processing time.

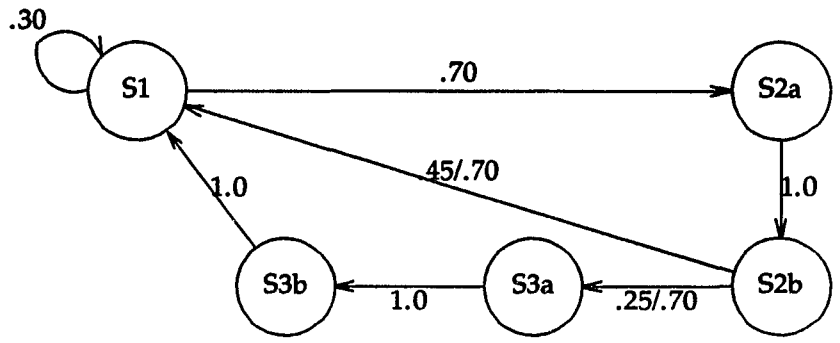


Figure 3.4: Cloning of states S2 and S3.

The arcs in figure 3.4 are now explained:

S1 -> S1: The arc from S1 to S1, labeled .30, represents the 30% of all memory access requests that utilize only one level of memory. The desired memory location exists in the highest level of memory, i.e., the memory level adjacent to the CPU.

S1 -> S2a: The arc from S1 to S2a, labeled .70, represents the 70% of all memory access requests that require more than one level of memory. The desired memory location does not exist in the highest level of memory, and at least one, if not more, additional levels of memory are needed.

S2a -> S2b: The arc from S2a to S2b, labeled 1.0, represents 100% (all) of memory level L2 requests that are delayed for two time units in memory level L2.

S2b -> S1: The arc from S2b to S1, labeled .45/.70, represents the 45% of all memory access requests that require exactly two levels of memory. These memory requests are delayed for exactly three units of time, one in memory level L1

and two in memory level L2. The desired memory location does not exist in the highest level of memory but does in the second level of memory.

S2b -> S3a: The arc from S2b to S3a, labeled .25/.70, represents the 25% of all memory access requests that require more than two levels of memory. The desired memory location does not exist in the first two levels of memory.

S3a -> S3b: The arc from S3a to S3b, labeled 1.0, represents 100% of all memory level L3 memory requests that are delayed for two time units in memory level L3.

S3b -> S1: The arc from S3b to S1, labeled 1.0, represents 100% of all memory level L3 requests that does not require any more levels memory, i.e., all memory requests that are not satisfied in memory level L1 or memory level L2, must be satisfied in memory level L3. These memory requests are delayed for exactly five units of time, one in memory level L1, two in memory level L2, and three in memory level L3.

When splitting (cloning) states, it is important that we have the appropriate number of arcs per path for all feasible paths. The actual delay is modeled by the number of arcs that a path through the memory system uses. Note that in figure 3.4. we may traverse three arcs in order to go from state S1 back to state S1; for example, we may traverse through states S2a and S2b, i.e., corresponding to a case when utilizing levels L2 and L1 only. These three arcs represent the two unit delay incurred in level L2 and the one unit delay incurred in level L1. The same reasoning applies for the five arcs traversed when utilizing level L3 (i.e., S1-S2a-S2b-S3a-S3b-S1.) Table 3.2. gives the state transition matrix associated with the new system model.

FROM	TO				
	S1	S2a	S2b	S3a	S3b
S1	0.3	0.7	0.0	0.0	0.0
S2a	0.0	0.0	1.0	0.0	0.0
S2b	.45/.70	0.0	0.0	.25/.70	0.0
S3a	0.0	0.0	0.0	0.0	1.0
S3b	1.0	0.0	0.0	0.0	0.0

Table 3.2: State-transition matrix for the model of figure 3.4.

The next-state probability equations of the new system model are given below:

$\hat{P}(S1)$ = .3 P(S1) + (.45/.7) P(S2b) + P(S3b)
$\hat{P}(S2a)$ = .7 P(S1)
$\hat{P}(S2b)$ = P(S2a)
$\hat{P}(S3a)$ = (.25/.7) P(S2b)
$\hat{P}(S3b)$ = P(S3a)

As before, P(Si) and $\hat{P}(Si)$ represent the current and next state probabilities respectively. The current-state probability of the system being in state S1 is .34483 as seen from the derivation that follows.

$\hat{P}(S1) + \hat{P}(S2a) + \hat{P}(S2b) + \hat{P}(S3a) + \hat{P}(3b)$
= .3 P(S1) + (.45/.7) P(S2b) + P(S3b)
+ .7 P(S1)
+ P(S2a)
+ (.25/.7) P(S2b)
+ P(S3a)

1.0 = .3 P(S1) + (.45/.7) (.7) P(S1)
+ (.25/.7) (.7) P(S1)
+ .7 P(S1)
+ .7 P(S1)
+ (.25/.7) (.7) P(S1)
+ (.25/.7) (.7) P(S1)

$$1.0 = 2.90\ P(S1)$$

$$P(S1) = .34483$$

We can now see from these results that our modified system, with an extra delay unit in levels L2 and L3, has an availability rating of 34.483% compared with the availability rating of 51.282% in the previous system. These results are consistent with what one might expect. The modified system has an extra delay in two levels of memory and thus a lower availability rating. A more realistic scenario would be possible by, perhaps, making some or all of the levels in the memory hierarchy function more efficiently, so that one or more of the memory levels is utilized less frequently. This added efficiency, for example, could be a better page replacement algorithm or possibility increasing the size of a memory level which in turn increases that level's hit ratio. The efficiency could also be in the form of changing a cache so instead of direct mapping to include fully associative, or better set associative mappings, thus making the given level less frequently accessed. Unfortunately this efficiency which might reduce the number of times a memory level is required, could also increase the delay encountered when a memory level is accessed.

For example, we examine a situation where more logic is added to level L3. This additional logic forces that level to take twice as long when it is processed, but on the other hand it may change the probabilities in such a way that it is needed in only 10% of all memory access requests as opposed to the 25% that was previously required. The rest of the old level L3 requests are now serviced in level L2. In conclusion, we now have the same 30% of all requests serviced in level L1, as before, 60% of the requests being serviced in level L2 and 10% of the requests serviced in level L3. Would this system be a better designed one from a system availability point of view? Figure 3.5 gives the state transition diagram of this newly modified system.

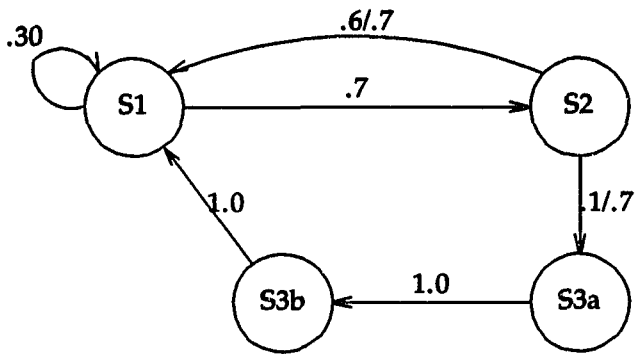


Figure 3.5: A modified 3 level memory hierarchy

The arcs in figure 3.5 are now explained:

S1 -> S1: The arc from S1 to S1, labeled .30, represents the 30% of all memory access requests that utilize only one level of memory. The desired memory location exists in the highest level of memory, i.e., the memory level adjacent to the CPU.

S1 -> S2: The arc from S1 to S2, labeled .7, represents the 70% of all memory access requests that require more than one level of memory. The desired memory location does not exist in the highest level of memory, and at least one if not more, additional levels of memory are needed.

S2 -> S1: The arc from S2 to S1, labeled .6/.7, represents the 60% of all memory access requests that requires exactly two levels of memory. The desired memory location does not exist in the highest level of memory but does in the second level of memory.

S2 -> S3a: The arc from S2 to S3a, labeled .1/.7, represents the 10% of all memory access requests that requires more than two levels of memory. The desired memory location does not exist in either of the first two levels of memory.

S3a -> S3b: The arc from S3a to S3b, labeled 1.0, represents 100% of all memory level L3 memory requests that is delayed for at least two time units in memory level L3.

S3b -> S1: The arc from S3b to S1, labeled 1.0, represents 100% of all memory level L3 requests that does not require any more levels of memory, i.e., all memory requests that are not satisfied in memory level L1 or memory level L2, must be satisfied in memory level L3. These memory requests are delayed for exactly four units of time, one in memory level L1, one in memory level L2, and two in memory level L3.

Table 3.3 below gives the state-transition matrix for this modified system as seen in figure 3.5.

FROM	TO			
	S1	S2	S3a	S3b
S1	0.3	0.7	0.0	0.0
S2	.6/.7	0.0	.1/.7	0.0
S3a	0.0	0.0	0.0	1.0
S3b	1.0	0.0	0.0	0.0

Table 3.3: State-transition matrix for the model of figure 3.5.

The next-state probability equations of the "next-state" / "current-state" relationship for the modified system are given below:

$\hat{P}(S1)$ = .3 P(S1) + (.6/.7) P(S2) + P(S3b)
$\hat{P}(S2)$ = .7 P(S1)
$\hat{P}(S3a)$ = (.1/.7) P(S2)
$\hat{P}(S3b)$ = P(S3a)

We can now solve for the current-state probability of the system being in state S1, noting that the sum of all "next-state" probabilities must be equal to one.

$$\hat{P}(S1) + \hat{P}(S2) + \hat{P}(S3a) + \hat{P}(S3b) = .3\ P(S1) + (.6/.7)\ P(S2) + P(S3b) + .7\ P(S1) + (.1/.7)\ P(S2) + P(S3b)$$

$$1.0 = .3\ P(S1) + (.6/.7)\ (.7)\ P(S1) + (.1/.7)\ (.7)\ P(S1) + .7\ P(S1) + (.1/.7)\ (.7)\ P(S1) + (.1/.7)\ (.7)\ P(S1)$$

$$1.0 = 1.90\ P(S1)$$

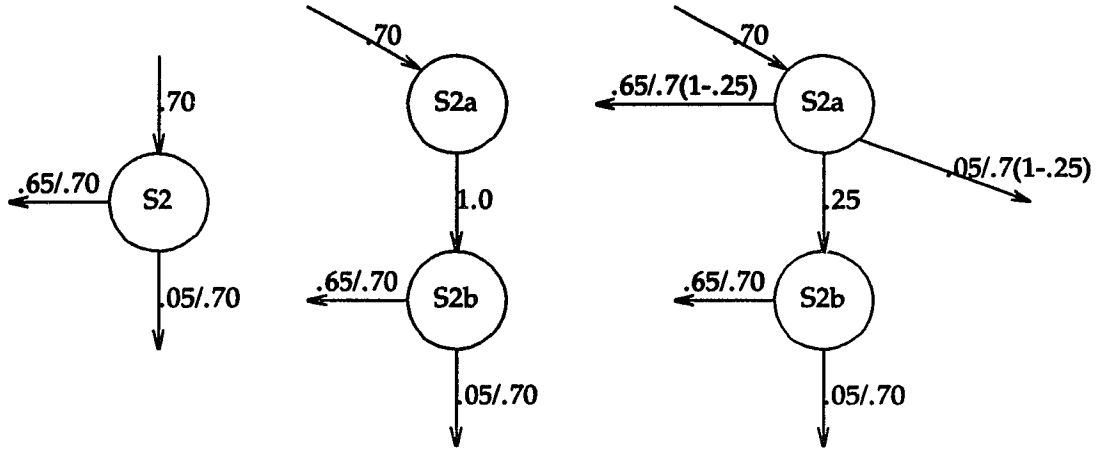
$$P(S1) = .52632$$

After solving the modified system for P(S1) we find that our system's availability rating is now 52.632%. This is an even better availability rating than the 51.282% for the original system. This is again consistent with what one would expect, since even though the system is incurring twice the delay in level L3 it uses L3 for less than half of the original level L3 requests.

3.5 Modeling Variable Delays

We now examine a potential design decision that could have been made during the building of the first model of the system we have analyzed. Assume that there is a possibility of adding more logic, e.g. in the form of perhaps a better page replacement algorithm that improves the hit ratio, in such a way that the number of requests that require level L3 is reduced by 80%, but each time that this level is required it may take as long as three cycles of processing time to satisfy the request. More precisely, it requires one,

two, and three cycles to service 50%, 25%, and 25% of level L3 requests, respectively. Unfortunately, in order to reduce the number of times that level L3 is needed, level L2 requires an additional cycle in 25% of all requests that need to utilize this level L2. Should we add this additional logic to the system? How much better, if at all, would this new and improved system be? In order to answer these questions one must either have a simulation or a more refined model as the one presented in this chapter. We start by building a state transition diagram with three states, one for each memory access level. These states are labeled: S1, S2, and S3, corresponding to levels L1, L2, and L3 respectively. State S3 must be cloned into three states S3a, S3b, and S3c to represent delays of one, two, and three processing cycles respectively as mandated by the proposed modification. State S2 must also be cloned into two states, as it may require one or two processing cycles. Figure 3.6 shows the cloning process for state S2. Figure 3.6a shows state S2 before cloning, and figure 3.6b shows state S2 cloned into two states with a fixed delay. Notice that all original entry points into state S2 enter through the first occurrence of the cloned S2 states, which is S2a, and all original exit points of state S2 leave from the last clone of that state, which is state S2b. For a fixed delay clone there is a single arc between clones with a probability of one. In a variable delay clone, the arc between clones could have a probability of less than one, but in these cases the excess of one over that probability must be made up in a prior state. In figure 3.6c we show that a probability of 0.25 connects state S2a to S2b, therefore the excess of 1 over 0.25, i.e., 1.0 - .25 = 0.75, must be made up in state S2a via exit arcs. It should be stressed that the excess must be distributed in exactly the same proportion as the original exit points of the state before cloning.



Before cloning **Figure 3.6a**

Fixed delay clone **Figure 3.6b**

Variable delay clone **Figure 3.6c**

State S3 of figure 3.6 is cloned in the same manner as state S2 of figure 3.6c, with the exception that state S3 is cloned into three states to represent a maximum delay of three. The probabilities assigned to the arcs from state S3 clones are now explained. The S3a to S1 arc represent the 50% of S3 requests that only require one cycle. The S3a to S3b arc represents the other 50% of S3 requests that require more than one cycle. The arc from S3b to S1 represents the 25% of all S3 requests that require exactly two cycles. This arc is given a probability of .5 (.5 of the S3b requests). Recall that S3b requests are .5 of S3a requests. This equates to .25 of the total S3 requests. The arc from S3b to S3c represent the 25% of S3 requests that require more than two cycles of processing. The S3c to S1 arc represents the S3c requests, they require only three cycles of processing. This arc from S3c to S1 is given a probability of 1.0 since three cycles is the maximum processing time that state S3 could use. Figure 3.7 gives the complete state-transition diagram for this new variable delay system.

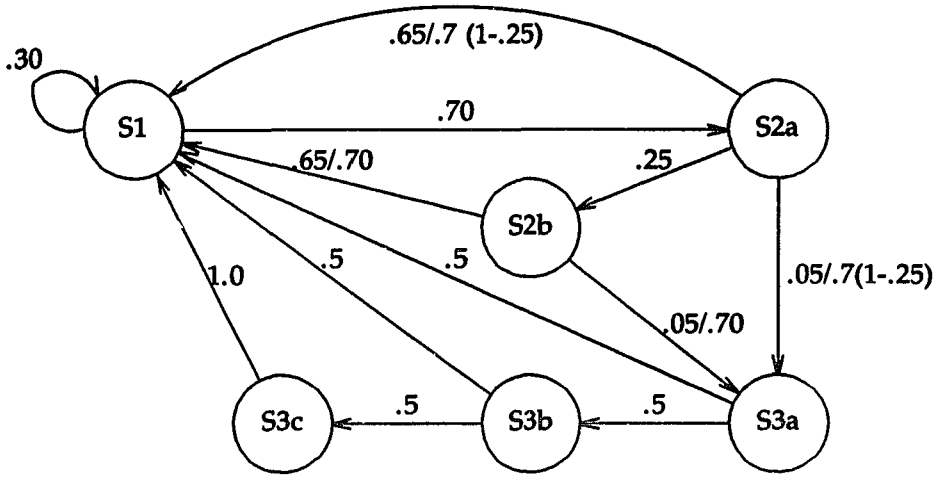


Figure 3.7: Variable Delay System

The arcs in figure 3.7 are now explained:

S1 -> S1: The arc from S1 to S1, labeled .30, represents the 30% of all memory access requests that utilize only one level of memory. The desired memory location exists in the highest level of memory, i.e., the memory level adjacent to the CPU.

S1 -> S2a: The arc from S1 to S2a, labeled .70, represents the 70% of all memory access requests that requires more than one level of memory. The desired memory location does not exist in the highest level of memory, and at least one, if not more, additional levels of memory are needed.

S2a -> S1: The arc from S2a to S1, labeled .65/.7(1-.25), represents the memory requests that only requires two levels of memory and encounters only one of two possible units of time delay in memory level L2. The probability for this arc has two components, the .65/.7 component represents the 65% of all memory requests that require exactly two levels of memory and the (1-.25) component

represents the 75% of memory level L2 requests that requires only one of two possible units of time delay. These memory requests are delayed for two units of time, one in memory level L1, and only one in the variable time delay memory level L2.

S2a -> S2b: The arc from S2a to S2b, labeled .25, represents the 25% of memory level L2 requests that requires two units of time for processing in memory level L2.

S2a -> S3a: The arc from S2a to S3a, labeled .05/.7(1-.25), represents the memory requests that requires more than two levels of memory and encounters only one of two possible units of time delay in memory level L2. The probability for this arc has two components, the .05/.7 component represents the 5% of all memory requests that require more than two levels of memory and the (1-.25) component represents the 75% of memory level L2 requests that requires only one of two possible units of time delay.

S2b -> S1: The arc from S2b to S1, labeled .65/.7, represents the memory requests that require exactly two levels of memory and encounter both units of time delay in memory level L2. The probability assigned to the arc, .65/.7 is that of the original probability, before cloning, of making a transition from S2 to S1. These memory requests are delayed for three units of time, one in memory level L1, and two in the variable time delay memory level L2.

S2b -> S3a: The arc from S2b to S3a, labeled .05/.7, represents the memory requests that require more than two levels of memory. The probability assigned to the arc, .05/.7 is that of the original probability, before cloning, of making a transition from S2 to S3.

S3a -> S1: The arc from S3a to S1, labeled .5, represents the 50% of memory level L3 requests that requires only one of three possible units of time delay.

S3a -> S3b: The arc from S3a to S3b, labeled .5, represents the 50% of memory level L3 requests that requires more than one of three possible units of time delay.

S3b -> S1: The arc from S3b to S1, labeled .5, represents the 25% (50% of 50%) of memory level L3 requests that requires exactly two of three possible units of time delay.

S3b -> S3c: The arc from S3b to S3c, labeled .5, represents the 25% (50% of 50%) of memory level L3 requests that requires more than two of three possible units of time delay.

S3c -> S1: The arc from S3c to S1, labeled 1.0, represents the 100% of memory level L3 requests that requires no more than three units of delay in memory level L3. This is the maximum delay that can be encounter in memory level L3.

The state-transition table for this system is given in Table 3.4.

FROM	TO					
	S1	S2a	S2b	S3a	S3b	s3c
S1	0.3	0.7	0.0	0.0	0.0	0.0
S2a	(.65/.70).75	0.0	.25	(.05/.70).75	0.0	0.0
S2b	(.65/.70)	0.0	0.0	(.05/.70)	0.0	0.0
S3a	0.5	0.0	0.0	0.0	0.5	0.0
S3b	0.5	0.0	0.0	0.0	0.0	0.5
S3c	1.0	0.0	0.0	0.0	0.0	0.0

Table 3.4: State-transition matrix for the model of figure 3.7.

The next-state probability equations of the variable time delay system are given below:

$\hat{P}(S1)$ = .3 P(S1) + (.65/.7)(.75) P(S2a) + (.65/.7) P(S2b)
+ .5 P(S3a) + .5 P(S3b) + P(S3c)
$\hat{P}(S2a)$ = .7 P(S1)
$\hat{P}(S2b)$ = .25 P(S2a)
$\hat{P}(S3a)$ = (.05/.7)(.75) P(S2a) + (.05/.7) P(S2b)
$\hat{P}(S3b)$ = .5 P(S3a)
$\hat{P}(S3c)$ = .5 P(S3b)

Each of the next-state probabilities can also be given in terms of the current-state probability of P(S1) as below:

$\hat{P}(S2a)$ = .7 P(S1)
$\hat{P}(S2b)$ = (.25)(.7) P(S1)
= .175 P(S1)
$\hat{P}(S3a)$ = (.05/.7)(.75)(.7) P(S1) + (.05/.7)(.25)(.7) P(S1)
= .05 P(S1)
$\hat{P}(S3b)$ = (.5)(.05) P(S1) = .025 P(S1)
$\hat{P}(S3c)$ = (.5)(.05)(.05) P(S1) = .0125 P(S1)

We can now solve for the current-state probability of the system being in state S1 as shown below:

$\hat{P}(S1) + \hat{P}(S2a) + \hat{P}(S2b) + \hat{P}(S3a) + \hat{P}(S3b) + \hat{P}(S3c)$
= .3 P(S1) + (.65/.7)(.75) P(S2a) + (.65/.7) P(S2b)
+ .5 P(S3a) + .5 P(S3b) + P(S3c)
+ .7 P(S1)
+ .25 P(S2a)
+ (.05/.7)(.75) P(S2a) + (.05/.7) P(S2b)
+ .5 P(S3a)
+ .5 P(S3b)

1.0 = .3 P(S1) + (.65/.7)(.75)(.7) P(S1)
+ (.65/.7)(.175) P(S1)
+ (.5)(.05) P(S1) + (.5)(.025) P(S1) + (.0125)P(S1)
+ .7 P(S1)
+ (.25)(.7) P(S1)
+ (.05/.7)(.75)(.7) P(S1) + (.05/.7)(.175) P(S1)
+ (.5)(.05) P(S1)
+ (.5)(.025) P(S1)

1.0 = 1.9625 P(S1)

P(S1) = .50955

After solving the for the current-state probability of the system being in state S1, we find that the availability rating is 50.955%. Thus, this last design does not perform as efficiently as the original one, and the modification was obviously a bad rather than a good design decision. If by chance, level L2 required an additional cycle in only 20%, as opposed to 25%, of the original level L2 memory requests, the availability rating of the system would be increased to 51.881%. The actual computation is left to the interested reader as an exercise. In this case, the modification could be a good design decision.

3.6 Summary

This chapter presents a novel approach that can act as an alternative or complement to simulation. This approach enjoys a definite improvement over simulation by allowing one to use analytical tools that pure simulation does not offer. Even though the discussion and examples pertained to memory systems, the model is by no means constrained to this application. This model can be used on a broad spectrum of similar problems with ease and similar success. A theoretical approach is superior to pure simulation, not only because it yields a closed-form solution but it also provides an alternative avenue of solution.

Chapter 4

Shared-Virtual-Memory

This chapter defines the concept of shared-virtual-memory in large scale parallel processing super-computers. Memory coherence in this context is also defined and algorithms for implementing shared-virtual-memory are given.

4.1 Overview

Most super-computers today are, in one form or another, a collection of many micro-computers. These micro-computers are usually housed in a single local physical unit with a high-speed communication channel which connects these micro-computers together. Each micro-computer works either independently or in parallel with other micro-computers of the system, in processing a sub-task of a possibly large problem. Each micro-computer is a completely independent computer, that in one possible configuration comes with its own private memory. The Intel IPC860 super-computer, for instance, is a collection of RISC type i860 micro-computers, connected together in a hypercube network. Each node in the hypercube is thus an independent computer which communicates with other nodes (computers) via exchanging communication messages. This type of parallel computer is commonly referred to as a Multiple Instruction Multiple

Data (MIMD) computer. Each processing element can run a different set of application instructions with different data sets (streams). In general, computers may be classified as belonging to one of four different categories [Flyn86]. These categories are 1) Single-Instruction Single-Data (SISD), 2) Single-Instruction Multiple-Data (SIMD), 3) Multiple-Instruction Single-Data (MISD), and 4) Multiple-Instruction Multiple-Data (MIMD).

(1) A SISD computer is a single processor that operates in a sequential manner on a single data stream. Computer instructions may be pipelined as well and some processing may be concurrently executed in different sites of the system. This pipeline processing is usually that of instruction fetch, instruction decode, and instruction execute. SISD is the most common form of computer architecture used today.

(2) SIMD computers process the same single instruction on multiple and different pieces of data. Each data is usually processed on a different processor. This type of computer is sometime referred to as an array processor. Each processing element, in the array, is under the control of a centralized single control unit. This control unit controls all processors, which in-turn executes a single instruction on different data structures.

(3) MISD computers also have many processing units, but this time many different instructions operate on the same single piece of data. The processors are usually set up as a pipe in the sense that the output data of one processor can become the input data of the next processor.

(4) MIMD computers have multiple processors as well, each one of which can operate independently of each other and on different data. About the only distinction between a MIMD computer and several SISD computers is in the communication links between the processors, the speed of the communication links, and the

method for loading data and programs into the processors. Most MIMD processors have a single resource manager which allocates resources or loads data and programs into the processors.

For more information on the architectural classification of computers the reader may refer to [Hwan84].

4.2 Software Development

The application development for MIMD computers requires the explicit decomposition of a problem into tasks that can be performed in parallel. The synchronization of these parallel tasks must also be explicitly developed for the particular application. Once this decomposition is accomplished these tasks must be coordinated via some form of inter-processor communication. For more information on the decomposing systems into modules the reader should refer to [Parn72]. Currently, on most MIMD computers, message passing is the only available form of inter-processor communication. The development of applications for parallel computers can be a difficult task. To simplify this task, shared memory can be used as an alternative form of inter-processor communication, as explained in the sequel.

Since each processor in this particular configuration has its own separate physical memory, the shared-memory-system must be implemented as virtual memory. In a traditional sense, virtual memory refers to memory which appears to be larger the the actual physical memory. The address space of main memory is extended through the use of secondary memory. This secondary memory is usually built around different (cheaper) technology such as disk memory. Unfortunately disk memory is substantially slower than main memory. The larger disk memory is copied into main memory, a page at a time upon request depending on the page replacement algorithm employed. This hopefully

allows most memory access to take place at the speed of the smaller but faster main memory and at a cost closer to that of the cheaper larger capacity disk memory. Shared-virtual-memory can be virtual from the traditional sense, of having a larger address space than the actual physical memory, and/or virtual from the "shared" sense. Being virtual from the shared sense means that memory, private to one processor, appears to be global memory to all other processors. Figure 4.1 gives an example of two CPU's each one with its own private memory, and figure 4.2 gives an example of two CPU's with a global shared memory.

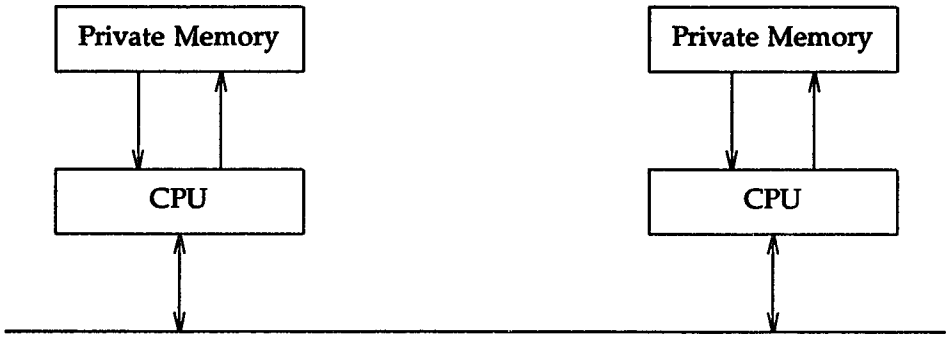


Figure 4.1: CPU's With Private Attached Memory

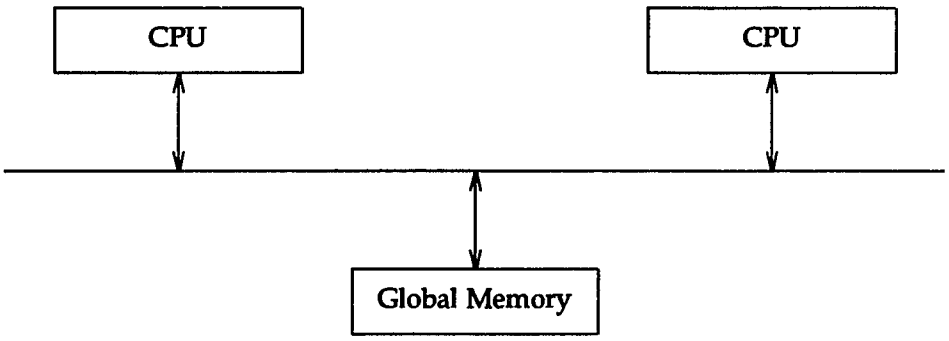


Figure 4.2: CPU's With Global Memory.

A shared-virtual-memory system transforms a system from an architecture similar to that of figure 4.1 into the behavior of a system whose architecture is that of figure 4.2. Even though the shared-virtual-memory system would use the message passing interface, this interface is naturally transparent to the programmer. The implementation chosen for a shared-virtual-memory system would affect the performance of an application. Several alternatives could be chosen when implementing a shared-virtual-memory system. This chapter presents some of those alternatives. The next chapter presents and explains a tool to choose the best implementation of a shared-virtual-memory system.

4.3 Memory Coherence

Memory is said to be coherent if the contents of a memory location returned from a read instruction is that of the most recent write to that location or, in other words, no stale data exists in the memory system. The implementation of shared-virtual-memory of a multiprocessor system must thus ensure memory coherency. Maintaining memory coherency could be a problem when multiple copies of the same memory location exist in different sets of processors each with its private attached local memory. This memory coherency problem may be even more complex if processors have their own private cache memories. If one instance of a memory location that resides in multiple memory banks is changed, the shared-virtual-memory system must insure that all other instances of that memory location are either updated or invalidated. One possible implementation of shared-virtual-memory that guarantees memory coherency is to allow only one instance of a memory location to exist. This approach, of a single instance of memory locations, could be accomplished by keeping the memory location fixed on a given processor or by letting the memory location migrate from processor to processor. A read or write operation in a single instance memory implementation first has to identify the physical memory

to which it should reference. Once the physical memory is identified, the actual operation is executed by the processor attached to that memory bank. Processor to processor communication would take place for read or write memory operations to memory locations that are not local. The communication overhead along with the additional CPU interrupts needed to perform a memory access associated with this implementation would be prohibitive. A more realistic implementation would be to allow memory locations to coexist on all processors as needed. In other words, to let every processor keep a copy of a desired memory location in its own local attached memory. This later implementation would be very similar to that of cache memory. In essence, each physical memory could be viewed as a cache or window into a much larger global memory. If a memory location can coexist in the attached local memory of multiple processors, then each write would require either a write to all current instances of that locations or invalidate all other copies and writing to the one local copy. The memory location might not exist locally, in which case a copy must be made before invalidation of other locations takes place.

4.4 Shared-Virtual-Memory

In a shared-virtual-memory system each page of memory must have a owner. It is the task of the owner to keep the page locally and send copies to other processors that request read access. If a processor requests write access to a page, the current owner must give up ownership of the page and pass the ownership to the requesting processor. When a new processor takes ownership of a page by virtue of a write operation, all other copies of the page must be invalidated. Ownership is transferred when a write operation is requested by a processor which is not the current owner. In order to maintain memory coherency, only one copy of a page can exist when write access is present. When read access is requested for a page which is currently reserved for writing, write access to that

page must be disallowed. The owner of the page, i.e., the processor who currently has the page with writing privileges, changes the access of the page to "read only". It is this "read only" copy which is given to the requesting processor. Now, as a result of the above explained scenario, multiple "read only" copies of the page exist, a copy for each requesting processor. One on the processor that originally had the write access, which was subsequently changed to "read only", and one copy on the processor that requested the "read" access. Note that many "read only" copies of the page can simultaneously exist while only one write copy of a page can exist at any given point in time. Whenever a write access is requested to a page, all other copies of that page must be invalidated. The task of distributing pages for read requests is the obligation of the owner. The task of keeping track of who currently owns a page is that of the shared-virtual-memory manager. The above manager must also keep track of which processors memory currently has a copy of the page, it is also the manager's job to invalidate pages when a write request is received.

4.5 Shared-Virtual-Memory Algorithms

In this section we examine shared-virtual-memory algorithms on loosely-coupled parallel processing systems. Processors are defined as being loosely-coupled if they do not physically share any memory but can access each others memory through interrupts or requests. Two processors physically share memory if they can both access a single memory location each through a single instruction. When two processors physically share memory they are considered to be tightly coupled. Most shared-virtual-memory algorithms for loosely-coupled parallel processing systems are based upon the concept described in the earlier section, where each page has an owner and manager. Algorithms for shared-virtual-memory differ mostly in the management of the pages. The manager

can be centralized, where one processor manages all pages. This type of algorithm is simple to implement but does not perform well and is susceptible to failure of the centralized manager. The reason for the poor performance of the centralized manager algorithm is due to massive amounts of communication traffic to and from the centralized manager. The centralized manager must also spend much CPU time keeping track of all pages in the system, which is another factor contributing to the reduced effectiveness of this algorithm. A better algorithm would thus be to have a distributed page manager. This way, communication traffic and page maintenance processing can be distributed across all processors in the system. A distributed page manager would also increase the robustness of the system by reducing the vulnerability to failure of the one single centralized processor. The distribution of the managers task can be done on a static or dynamic basis. In a static implementation, each processor is given a predetermined set of pages to manage. When a processor executes a read or write instruction for a memory location that does not currently exist in local attached memory, a page fault occurs. The page fault handler determines which processor is currently acting as the manager of the page and then requests that page from the manager. The manager would then relay this request to the owner of the page for proper response.

4.6 An Example Shared-Virtual-Memory Algorithm

In this section a shared-virtual-memory algorithm is developed. This algorithm is used for developing the shared-virtual-memory performance model in chapter 5. Two sets of data structures are needed, one set exists on all processors in the system and the other set only exists on page manager processors. The data structure maintained by the page manager store the current owner of a page, the current set of processors that have a copy of the page, and a lock field for synchronizing the access to this structure. The page

manager has one of these structures for every page of shared-virtual-memory that it manages. Each processor in the system maintains a data structure for each page of shared-virtual-memory that currently resides in its local attached memory. This data structure contains the allowed access of the corresponding page, and a lock field for synchronizing the access to this structure. The access for a page can either be "read-only", "read-write", or "invalid". The invalid access could also be viewed as a free entry in the page table. If there is a physical page of memory for every entry in the page table, this invalid access code can also be viewed as a marker for a free page of memory.

The actual details of the algorithm is given in four separate parts; the "read fault handler", the "page manager", the "page server", and the "write fault handler". Each of these separate but interacting parts are explained below:

(1) *The Read Fault Handler.*

A read fault handler resides on every processor in the shared-virtual-memory system. When a read fault occurs, i.e., a processor executes a read instruction for a memory location that does not currently exist in local attached memory, the read fault handler obtains a copy of the page with the desired memory location. This is accomplished by contacting the manager of the required page and requesting a read-only copy of the page. When a read-only copy of the page is obtained, the page is marked read-only in the page table of the faulting processor.

(2) *The Page Manager.*

There is one and only one page manager for each page of shared-virtual memory. There may be one page manager for all pages in the system, i.e., centralized manager implementation, or in a distributed implementation there may be multi-

ple page managers, where each page manager manages a group or set of pages. Different pages may be managed by page managers residing on different processors. When a page manager receives a request for a read-only page, i.e., a read fault has occurred, the page manager updates the data structure which contains a list of all processors which have a copy of the page to include the new requesting processor. A message is then sent by the page manager to the owner of the page. This messages directs the owner of the page to mark the page as "read-only" in the owner's page table and to send a copy of that page to the requesting processor. The page manager always know which processor is currently the owner of a page and which processors currently have copies of a page. If the page was requested for writing, i.e., a write fault has occurred, the page manage sends a page invalidation message to each processor which currently has a copy of the page. The page manager then updates its data structure to set the requesting processor as the new owner and also set the list of processors which currently have a copy of the page to be only the owner.

(3) *The Page Server*

The page server for a particular page resides on the processor which currently owns the page and it is the page servers function to distribute pages when needed. Requests for pages made by other processors are usually received from the manager of a particular page, to send the mentioned page to a requesting processor. If the processor is requesting the page for reading, the server marks the page as read-only in its own page table and sends out a copy of the page. If the request is for a read-write page, the server marks the page locally as invalid and sends out a copy of the page. For read-write pages, the server relinquishes control of the page and the requesting processor now owns the page and becomes the

new page server for that page.

(4) *The Write Fault Handler.*

The write fault handler, like the read fault handler, resides on every processor of the shared-virtual-memory system. The write fault handler is called upon when a processor executes a write instruction for a memory location that either does not currently exist in local attached memory, or the page in which the memory location resides is there but marked as read only. In either case, a request must be made to the manager of the page. When the page is requested for writing, the requesting processor becomes the new owner of the page.

As stated earlier, there could be one or many page managers. If there were only one centralized page manager which resides on one processor in the system, there would be much communication traffic to and from this processor. This single page manager processor would also be penalized by having less processing time for jobs other than page-maintenance, while maintaining all pages in the system. For these two reasons a single centralized page manager is not the most efficient solution for maintaining cache coherency. It also suffers from a problem of all centralized systems which is the vulnerability to failure of the centralized controller, any such failure of the centralized controller is catastrophic. In any event, a more efficient page management method would be to distribute the task of the page manager across multiple processors in the system.

4.7 A More Efficient Page Manager

Shared-virtual-memory algorithms can be implemented with a centralized or distributed page manager. Centralized page management algorithms have insufficiencies, such as those explained in the previous section, which limit there usefulness. Distributing the

function of the page manager provides a more efficient utilization of the overall resources in the system. Distributed page managers can be implemented on a static or dynamic basis. Static page managers have the pages which they manage defined during system initialization. These pages remain the responsibility of that manager until they are manually changed and the system is re-initialized. Each processor, during the initialization phase in the shared-virtual-memory system, receives a mapping of pages to page managers. This mapping is used for a table look-up when a processor encounters a memory access fault. Static page assignments may perform well for one application but poorly for another. If the pages are assigned to processors for management dynamically then the page assignments can be constantly adjusted. This allows the algorithm to perform well for a larger set of applications.

A dynamic distributed page manager can be implemented with only minor modifications to the static distributed system. These modifications address the task of how to find the page manager. One algorithm that could be used for finding page managers, is to start as if it were static and update tables each time a page manager is changed. This method could require many unused table updates. A modification to this algorithm would be to only update tables when access to the page manager is needed. This would eliminate the unused table updates, but add additional delays when first sending a request to a page manager. In the next chapter these additional delays, which are variable in nature, are discussed in detail.

One last performance improvement to the dynamic distributed page manager algorithm is to move the task of the page manager to the owner. In the static or centralized algorithms, each processors always knew the location of the page manager for every page and the page manager always knew the location of the owner. There was a definite need for the page manager, i.e., to find the owner. We now find the processors searching for

the page manager, why not let them search for the owner? Combining the page manager and page owner functions provides for the most efficient general purpose algorithm.

Chapter 5

Performance Analysis Of Shared-Virtual-Memory

This chapter presents the necessary methodology for analytically analyzing the performance of shared-virtual-memory in large scale parallel processing super-computers.

5.1 Shared-Virtual-Memory Access: Read Or Write

The first part of this chapter examines the shared-virtual-memory algorithm given in section 4.6 of the previous chapter. The read and write operations are examined separately with the examination of the read operation first. The distinction between a read and write operation is that read operation can always function if the page that contains the desired address is present in a processor's local attached memory. This is true regardless of whether the page is "read-only" or "read-write". On the other hand, for a write operation to take place, the page which contains the desired write location must have "read-write" access, as opposed to just the mere existence of a page in the processor's local attached memory for a successful read operation.

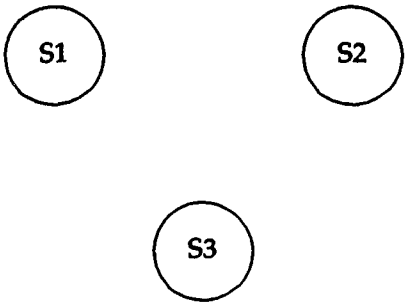
5.2 A Model For A Shared-Virtual-Memory Read

When a read fault occurs, i.e., the desired memory location is not currently present in the processor's local attached memory, the faulting processor, i.e., the processor creating this fault, must request a copy of the page from the page manager. It should be noted, the faulting processor could also be the page manager, in which case the request to the page manager would be trivial. This request is considered a trivial request since it would not require any inter-processor communication to take place. Once the page manager receives the request for the page, it must ascertain which processor is currently the required owner. When the owner processor is determined, the page manager forwards the request for a copy of the page, to the page owner. The owner of the page then sends a copy to the faulting processor. The owner of the page always has a copy of the page, thus the faulting processor could not be the owner of the page. Although, in a write operation a processor can own a read-only page, and thus encounter a fault.

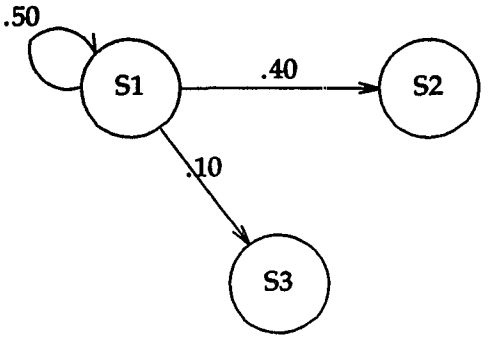
To better understand the model that we use to evaluate the performance of a shared-virtual-memory read, we assume some particular statistics for illustrative purposes only. Assume therefore, that 50% of all read requests are satisfied without a read fault, i.e., these requests are for the contents of locations that currently reside in the processor's local attached memory. Let us further assume that 10% of all read requests would cause a read fault to occur and the faulting processor is the manager of the page that contains the desired memory location. The remaining 40% of all read requests would occur due to locations which do not currently reside in the processors attached memory and for which the processor is not the manager of the page containing the desired location. There is also the possibility that the manager of the page with the desired memory location is also the owner of the page, but not the faulting processor. Assume that the manager and the owner coincide for 10% of all read faults in which the faulting processor is not the

manager. The state-transition diagram is now constructed for this read instruction. Figure 5.1.a gives the states of shared-virtual-memory read request: state S1 represents a read request for shared-virtual-memory and is the initial state that all requests for shared-virtual-memory must go through. State S2 represents a read fault from a processor that is not the manager of the page containing the desired location. This state (S2) is utilized when a faulting processor sends a request, for the page that contains the desired location, to the page manager. State S3 represents the owner of a page, when the owner and the manager reside on different processors. Figure 5.1b shows all possible transitions from state S1. The arc from S1 to S1 labeled .50 represents the 50% of all read requests that can be satisfied without a read fault, these requests are for the contents of locations that currently reside in the processor's attached memory. The arc from S1 to S2, labeled .40 represents the 40% of all read requests for locations which do not currently reside in the processors attached memory, and for which the processor is not the manager of the page containing the desired location. The arc from S1 to S3, labeled .10 represents the 10% of all read requests that cause a read fault and the faulting processor is the manager of the page that contains the desired memory location. Figure 5.1c shows all possible transitions from state S2: the arc from S2 to S1, labeled .10, represents the 10% of all requests to the page manager in which the page manager is also the owner. In this case the page can be sent back to the faulting processor and the read can be satisfied. The arc from S2 to S3 labeled ".90" represents the remaining 90% of read faults in which the faulting processor is not the manager of the page containing the desired memory location and the manager is not the owner. In this case the manager must send a request to the owner of the page for a copy of the page and this page, in turn, is given to the requesting processor. Figure 5.1d shows all possible transitions from state S3; state S3 represents the owner of the page containing the desired memory location which is not the manager, and that the manager has informed the owner that the page is needed. The owner must always have a

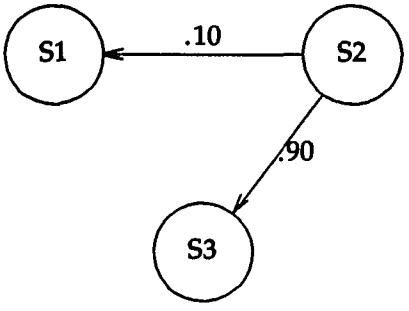
copy of the page and therefore once in state S3-the only move is back to state S1; this arc is labeled "1.0" since this is the probability associated with that transition from state S3 back to state S1. Figure 5.1e gives the complete and final state-transition diagram associated with this read instruction.



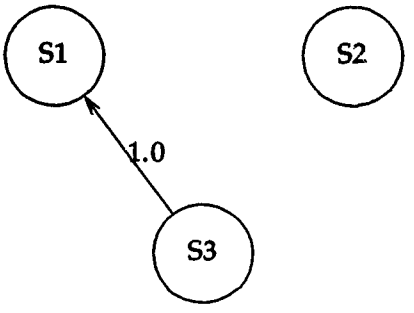
States of the model
Figure 5.1a



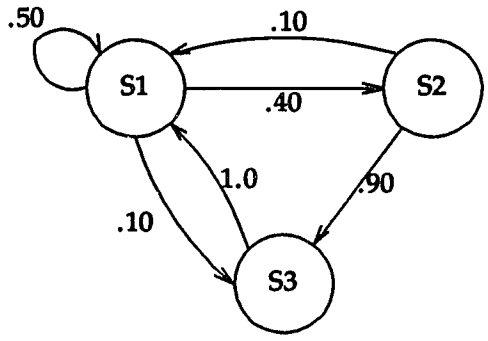
Transitions from state S1
Figure 5.1b



Transitions from state S2
Figure 5.1c



Transitions from state S3
Figure 5.1d



State-transition diagram of a shared-virtual-memory read.
Figure 5.1e

The arcs in figure 5.1e are now explained:

S1 -> S1: The arc from S1 to S1, labeled .50, represents the 50% of all read requests that are satisfied without a read fault. The desired memory location exists in the processor's local attached memory.

S1 -> S2: The arc from S1 to S2, labeled .40, represents the 40% of all read requests in which a read fault occurs and the faulting processor is not the manager of the page which contains the desired memory location.

S1 -> S3: The arc from S1 to S3, labeled .10, represents the 10% of all read requests in which a read fault occurs and the faulting processor is the manager of the page which contains the desired memory location.

S2 -> S1: The arc from S2 to S1, labeled .10, represents 10% of all read faults in which the faulting processor is not the manager of the page which contains the desired memory location. This situation occurs when the manager of the page is also the owner of the page which contains the desired memory location. The arc returns to state S1 because the page has been found and the read request can now be satisfied.

S2 -> S3: The arc from S2 to S3, labeled .90, represents 90% of all read faults in which the faulting processor is not the manager of the page which contains the desired memory location. This situation occurs when the owner of the page is not the manager. The manager must forward the request to the processor which currently owns the page.

S3 -> S1: The arc from S3 to S1, labeled 1.0, represents 100% of all read faults in which the faulting processor is not the manager of the page, and the manager is not the owner of the desired page. State S3 represents the owner of the page. The arc returns to state S1 because the page has been found and the read request can now be satisfied.

After constructing the state-transition diagram (see figure 5.1e) the next step in analyzing the shared-virtual-memory read instruction is to build a corresponding state-transition matrix. This matrix is constructed in the same manner as in the performance model of chapter 3 section 3.2 and is given below in Table 5.1.

FROM	TO		
	S1	S2	S3
S1	0.5	0.4	0.1
S2	0.1	0.0	0.9
S3	1.0	0.0	0.0

Table 5.1: State-transition matrix for the model of figure 5.1e.

After the state-transition matrix is constructed, the read availability is now computed with the same methodology as given earlier in chapter 3 (section 3.3). The next-state probability equations of the shared-virtual-memory read are given below:

$\hat{P}(S1)$ = .5 P(S1) + .1 P(S2) + P(S3)
$\hat{P}(S2)$ = .4 P(S1)
$\hat{P}(S3)$ = .1 P(S1) + .9 P(S2)

The read-availability of the system is .53763 as seen from the derivation that follows.

$\hat{P}(S1) + \hat{P}(S2) + \hat{P}(S3)$ = .5 P(S1) + .1 P(S2) + P(S3)
+ .4 P(S1)
+ .1 P(S1) + .9 P(S2)

1.0 = .5 P(S1) + (.1) (.4) P(S1) + .1 P(S1)
+ (.9) (.4) P(S1)
+ .4 P(S1)
+ .1 P(S1) + (.9)(.4) P(S1)

1.0 = 1.86 P(S1)

P(S1) = .53763

5.3 A Model For A Shared-Virtual-Memory Write

As stated in section 5.1, a write instruction in this protocol behaves differently than a read instruction. In a read instruction, if the page is found in a processor's local attached memory, it can be used. The write instruction requires, on the other hand, that the page be marked for writing or labeled read-write. When a page is labeled read-write, no other copies of the page may exist. This also implies that the processor with the page marked for read-write is the owner of the page. When a processor encounters a write operation to a shared-virtual-memory location, that location must exist locally and be marked read-write. If both are not the case a write fault occurs. If the processor is not already the owner of the page, it must become the owner of that page in order to complete the write instruction. Since only one processor can be the owner of a page at any given time, only that one processor can write to that particular page, at that point in time,

thus coherency is maintained. If a processor is not the owner of the requested page, the processor must find the owner by requesting the page from the manager. It should be noted that a processor that owns a page can still encounter a write fault if the page is marked read-only. This last situation occurs if other copies of the page currently exist on other processors. When multiple copies of a page exist, a read-write copy may not exist. A write would require the resulting processor to become the owner of the page, and at the same time, invalidate all other copies of that page. When only one copy of the page exists, that page could then be marked "read-write" and the write can now take place. To better understand the write model, some particular statistics are assumed for illustrative purposes only.

Assume that 25% of all write requests are satisfied without a write fault, these requests are thus for the contents of memory locations that currently reside in the processor's attached local memory and in a page that is marked "read-write". The remaining 75% of all write requests therefore causes a write fault to occur. When a write fault occurs, it can be classified into one of four different categories which are described below:

(1) The faulting processor is not the owner of the page and is also not the manager of the page. In this category, a request for write access to a page would be made from the faulting processor to the manager of the page. For this example it is assumed that this scenario is the case for 60% of all write instructions.

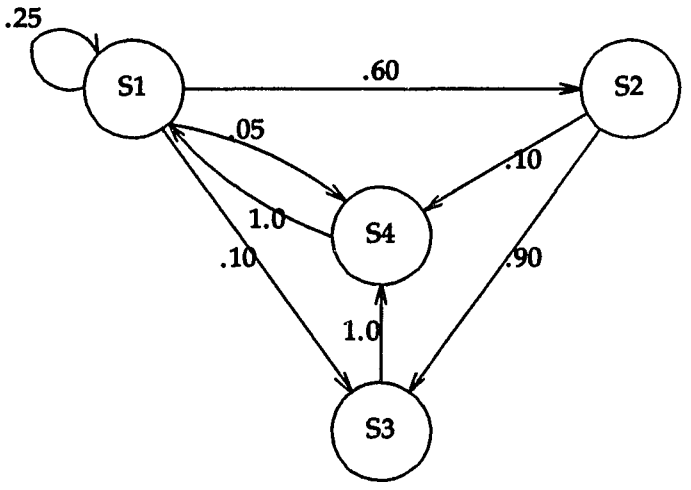
(2) The faulting processor is the manager, but not the owner, of the page which contains the desired location for writing. In this case the faulting processor would know who the owner of the page is, along with where any read-only copies of the page might be. In order to perform the write instructions, the faulting processor requests ownership from the owner and invalidates all read-only copies of the

page. This scenario occurs for 10% of all write instructions.

(3) This is the case where the faulting processor is the manager and the owner of the page which contains the desired location for writing. This situation implies that the faulting processor currently has the page as "read-only". Since the processor is also the owner of the page, it must have a copy of the page, and this copy must be "read-only", otherwise a write fault would not have occurred. In this case all other "read-only" copies of the page must be invalidated before this page can be changed to "read-write". It is assumed that this occurs for 5% of all write faults.

(4) The faulting processor in this case is the owner, but not the manager, of the page which contains the desired location for writing. In this situation, since a write fault has occurred, the faulting processor must currently have a "read-only" copy of the page, this is for the same reasons as in case 3 above. In this case, the manager of the page would be asked to invalidate all other "read-only" copies of the page so the faulting processor can have a "read-write" access to the requested page. This situation would cause the same chain of events to take place as mentioned earlier in case 1, for this reason we consider this case to be the same as that of case 1 above and consider these events to be included in the 60% write faults that fall into that case.

Figure 5.2, gives the state transition diagram for a shared-virtual-memory write instruction.



State-transition diagram of a shared-virtual-memory write
Figure 5.2

The arcs in figure 5.2 are now explained:

S1 -> S1: The arc from S1 to S1, labeled .25, represents the 25% of all write instructions that are satisfied without a write fault. The page of memory which contains the desired location exists in the local processor's attached memory and is "read-write".

S1 -> S2: The arc from S1 to S2, labeled .60, represents the 60% of all write instructions that have encountered a write fault and the faulting processor is not the manager of the page in fault.

S1 -> S3: The arc from S1 to S3, labeled .10, represents 10% of all write instructions. These write instructions have encountered a write fault, and the faulting processor is the manager but not owner of the page containing the desired location for writing.

S1 -> S4: The arc from S1 to S4, labeled .05, represents 5% of all write instructions. These write instructions have encountered a write fault and the faulting pro-

cessor is both the manager and owner of the page containing the desired location for writing.

S2 -> S3: The arc from S2 to S3, labeled .90, represents 90% of write faults in which the faulting processor is not the manager of the page in fault. These 90% of all S1 -> S2 transitions are the write faults that occur when the page manager is not the owner and not the faulting processor.

S2 -> S4: The arc from S2 to S4, labeled .10, represents the remaining 10% of the write faults for which the faulting processor is not the manager of the page in fault. These 10% of all S1 -> S2 transitions are the write faults that occur when the page manager and the owner of the page reside on the same processor which is not the faulting processor.

S3 -> S4: The arc from S3 to S4, labeled 1.0, represents 100% of all write instructions that encounter a fault, and the owner of the page is not the manager of the page or the faulting processor. This arc represents the situation that once the owner is located, all read-only pages in the system must be invalidated. State S4 represents the invalidation of pages.

S4 -> S1: The arc from S4 to S1, labeled 1.0, represents 100% of all write instructions that have encountered a fault and have completed the invalidation of all other pages that contained the desired write location in the system. From this point we always return to the faulting processor to complete the write operation.

After constructing the state-transition diagram (see figure 5.2) the next step in developing the protocol for analyzing the shared-virtual-memory write instruction is to build the corresponding state-transition matrix. The state-transition matrix is constructed in the same manner as in the read model for the shared-virtual-memory of the previous

section and is given below in Table 5.2.

FROM	TO			
	S1	S2	S3	S4
S1	0.25	0.60	0.10	0.05
S2	0.00	0.00	0.90	0.10
S3	0.00	0.00	0.00	1.00
S4	1.00	0.00	0.00	0.00

Table 5.2:

State-transition matrix for a Shared-virtual-memory write.

After the state-transition matrix is constructed, the shared-virtual-memory write availability can now be computed employing the same methodology as was used earlier for computing the availability of the read operation in shared-virtual-memory. The next-state probability equations of the write operation for the shared-virtual-memory are given below:

$$\hat{P}(S1) = .25\ P(S1) + P(S4)$$
$$\hat{P}(S2) = .60\ P(S1)$$
$$\hat{P}(S3) = .10\ P(S1) + .90\ P(S2)$$
$$\hat{P}(S4) = .05\ P(S1) + .10\ P(S2) + P(S3)\ .$$

The shared-virtual-memory write availability of the system is .33445, as seen from the derivation that follows.

$$\begin{aligned}\hat{P}(S1) + \hat{P}(S2) + \hat{P}(S3) + \hat{P}(S4) = {} & .25\ P(S1) + P(S4) \\ & + .60\ P(S1) \\ & + .10\ P(S1) + .90\ P(S2) \\ & + .05\ P(S1) + .10\ P(S2) + P(S3)\end{aligned}$$

$$
\begin{aligned}
1.0 = \; & .25\,P(S1) + .05\,P(S1) + (.10)(.60)\,P(S1) \\
& + .10\,P(S1) + (.90)(.60)\,P(S1) \\
& + .60\,P(S1) \\
& + .10\,P(S1) + (.90)(.60)\,P(S1) \\
& + .05\,P(S1) + (.10)(.60)\,P(S1) \\
& + .10\,P(S1) + (.90)(.60)\,P(S1)
\end{aligned}
$$

$$1.0 = 2.99\,P(S1)$$

$$P(S1) = .33445$$

The model described in this section makes some implicit assumptions about the time it consumes in each state, it assumes that all states take the same unit of time. It also assumes that a single function always take the same amount of time each time it is invoked. For example, examine the page invalidation function which must invalidate all pages in the system, it could take substantially longer if the protocol had to invalidate many pages as opposed to one. The next section gives a more detailed examination of the invalidate function.

5.4 A Detailed Look At The Invalidate Function

The shared-virtual-memory invalidation function must ensure that all copies, except one, of the requested shared page are invalidated. This invalidation is required in order to maintain cache coherency when a write fault occurs. The invalidation protocol ensures that each processor which has a copy of the page, for which the write fault has occurred, removes or invalidates its copy of that page. After invalidation has taken place, only one copy of the page exists. The one remaining copy can now be safely modified. In our example of a shared-virtual-memory protocol, as given in section 4.6, invalidation was performed by the page manager. The page manager keeps track of every processor that has a copy of a shared-virtual-memory page. The name of each processor which has a

copy of a shared-virtual-memory page is kept in the list field of the page manager's data structure for that page. If this page is requested for writing, the page manager uses the list field in the data structure for that page, to send invalidation messages. To ensure that the invalidation process is complete, an invalidation completed response must be received from each of the processors to which an invalidation message was sent out. Once all invalidation completed messages have been received, a write to that particular page could then take place.

The invalidation task consumes a variable amount of time, depending upon the number of processors that invalidation messages must be sent to. In the model for a write operation into a shared-virtual-memory, as presented in section 5.3, the invalidation time is always assumed to be one unit. In the next section the shared-virtual-memory write performance model includes a variable time page invalidation function.

5.5 A Variable Time Invalidation Function

In this section the performance of a write operation of a shared-virtual-memory model is expanded to handle variable delay invalidation. The variable delay invalidation is modeled using the variable delay state cloning techniques presented earlier in chapter 3. To illustrate the variable delay cloning, it is assumed that the delay encountered by the page invalidation protocol can be of one, two, or three time units. These delay units are chosen for illustrative purposes only and correspond to invalidation operations that take a short amount of time, a medium amount of time, and a long amount of time, respectively. The actual amount of time require by the invalidation protocol's operation is dictated by the amount of processors that have copies of the page which is invalidated. Let us assume for the purpose of demonstrating the technique that 20% of all page invalidation operations take place in one unit of time, 60% of all page invalidation operations take

place in two units of time, and the remaining 20% of all page invalidation operations take place in three units of time. Figure 5.3 below shows the entry and exit arcs corresponding to state S4 of figure 5.2, which represents the invalidation before state cloning.

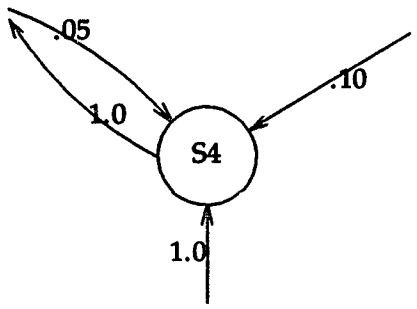


Figure 5.3: Invalidation State S4 of Figure 5.2 Before Cloning

State S4 of Figure 5.3 above is cloned into three states, one for each of the three delay units assumed for the invalidation protocol. The three states labeled S4a, S4b, and S4c represent a delay of one, two, and three units of time, respectively. Before cloning, state S4 has three incoming arcs and one outgoing arc. The incoming arcs are all directed toward state S4a, the first in the chain of new states. Figure 5.4 below shows the three new states S4a, S4b, and S4c that replace the single invalidation state S4 in figure 5.5.

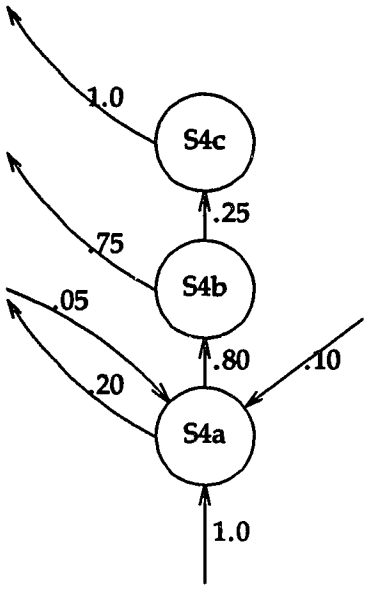
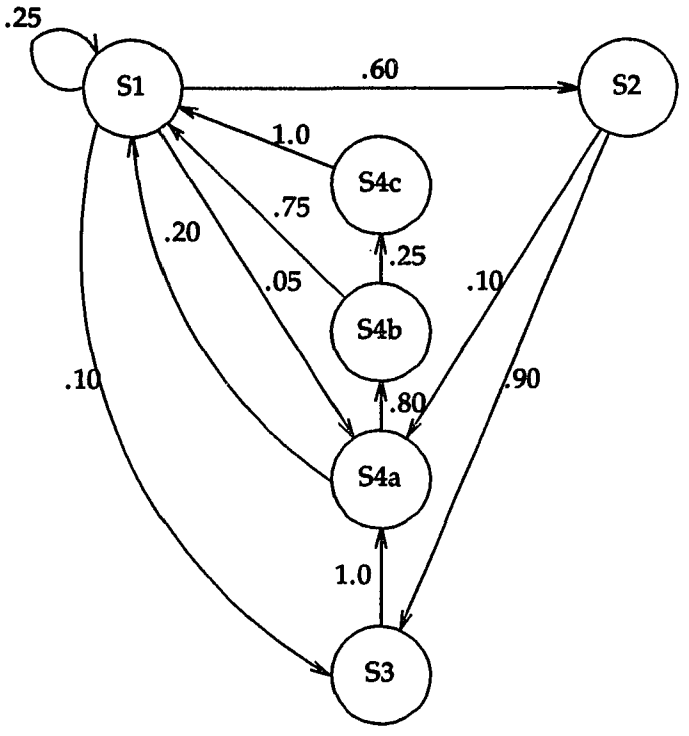


Figure 5.4: Invalidation State After Cloning

In figure 5.4, the arc from state S4a, labeled .20, represents the 20% of all page invalidation operations that takes place in one unit of time. The arc from state S4a to state S1, labeled .80, represents the 80% of all page invalidation operations that take longer than one unit of time. The arc from state S4b to state S1, labeled .75 represents the 60% (75% of 80%) of all page invalidation operations that take two units of time. The arc from state S4b to state S4c, labeled .25, represents the 20% (25% of 80%) of all page invalidation operations that take longer than two units of time. The final arc from state S4c to state S1, labeled 1.0, represents the 20% (25% of 80%) of all page invalidation operations that takes exactly three units of time. Figure 5.5 below shows the complete state-transition diagram for a write protocol into a shared-virtual-memory with a variable delay invalidation function.



State-transition diagram of a variable invalidate write protocol.
Figure 5.5

The arcs in figure 5.5 are now explained:

S1 -> S1: The arc from S1 to S1, labeled .25, represents the 25% of all write instructions that are satisfied without a write fault. The page of memory which contains the desired location exists in the local processor's attached memory and is "read-write".

S1 -> S2: The arc from S1 to S2, labeled .60, represents the 60% of all write instructions that have encountered a write fault and the faulting processor is not the manager of the page in fault.

S1 -> S3: The arc from S1 to S3, labeled .10, represents 10% of all write instructions. These write instructions have encountered a write fault, and the faulting pro-

cessor is the manager but not owner of the page containing the desired location for writing.

S1 -> S4a: The arc from S1 to S4a, labeled .05, represents 5% of all write instructions. These write instructions have encountered a write fault and the faulting processor is both the manager and owner of the page containing the desired location for writing.

S2 -> S3: The arc from S2 to S3, labeled .90, represents 90% of write faults which the faulting processor is not the manager of the page in fault, These 90% of all S1 -> S2 transitions are the write faults that occur when the page manager is not the owner and not the faulting processor.

S2 -> S4a: The arc from S2 to S4a, labeled .10, represents the remaining 10% of the write faults for which the faulting processor is not the manager of the page in fault. These 10% of all S1 -> S2 transitions are the write faults that occur when the page manager and the owner of the page reside on the same processor which is not the faulting processor.

S3 -> S4a: The arc from S3 to S4a, labeled 1.0, represents 100% of all write instructions that encounter a fault, and the owner of the page is not the manager of the page or the faulting processor. This arc represents the situation that once the owner is located, all read-only pages in the system must be invalidated. States S4a, S4b, and S4c represent the invalidation of pages.

S4a -> S1: The arc from S4a to S1, labeled .20, represents 20% of all write instructions that have encountered a fault and have just completed the invalidation in one of three possible time units.

S4a -> S4b: The arc from S4a to S4b, labeled .80, represents 80% of all write instructions that have encountered a fault and require more than one time unit for

invalidation.

S4b -> S1: The arc from S4b to S1, labeled .75, represents 60% (75% of 80%) of all write instructions that have encountered a fault and have just completed the invalidation in exactly two of three possible three time units.

S4b -> S4c: The arc from S4 to S1, labeled .25, represents 20% (25% of 80%) of all write instructions that have encountered a fault and require more than two time units for invalidation.

S4c -> S1: The arc from S4 to S1, labeled 1.0, represents 100% of all write instructions that have encountered a fault and have just completed the invalidation in exactly three of three possible time units.

The state-transition diagram given in figure 5.5 is the result of replacing state S4 in figure 5.2 with the variable delay cloned states of figure 5.4. We now have a shared-virtual-memory write model with a variable delay in the page invalidation protocol function. The next step in analyzing the performance of this new and improved shared-virtual-memory write system is to construct the state-transition matrix. Table 5.3 below contains that matrix.

FROM	TO					
	S1	S2	S3	S4a	S4b	S4c
S1	0.25	0.60	0.10	0.05	0.00	0.00
S2	0.00	0.00	0.90	0.10	0.00	0.00
S3	0.00	0.00	0.00	1.00	0.00	0.00
S4a	0.20	0.00	0.00	0.00	0.80	0.00
S4b	0.75	0.00	0.00	0.00	0.00	0.25
S4c	1.00	0.00	0.00	0.00	0.00	0.00

Table 5.3: State-transition matrix for a Shared-virtual-memory variable write.

After the state-transition matrix is constructed, the shared-virtual-memory write availability can now be computed with the same methodology as the one used in the previous section. The next-state probability equations of the shared-virtual-memory write protocol with variable page invalidation delay are given below:

$\hat{P}(S1)$ = 0.25 P(S1) + 0.20 P(S4a) + 0.75 P(S4b) + P(S4c)
$\hat{P}(S2)$ = .60 P(S1)
$\hat{P}(S3)$ = .10 P(S1) + .90 P(S2)
$\hat{P}(S4a)$ = .05 P(S1) + .10 P(S2) + P(S3)
$\hat{P}(S4b)$ = .80 P(S4a)
$\hat{P}(S4c)$ = .25 P(S4b)

The shared-virtual-memory write availability of the system with a variable page invalidate delay is .26738, as seen from the derivation, for the state S1, that follows.

$\hat{P}(S1) + \hat{P}(S2) + \hat{P}(S3) + \hat{P}(S4a) + \hat{P}(S4b) + \hat{P}(S4c)$
= .25 P(S1) + .20 P(S4a)
+ .75 P(S4b) + P(S4c)
+ .60 P(S1)
+ .10 P(S1) + .90 P(S2)
+ .05 P(S1) + .10 P(S2) + P(S3)
+ .80 P(S4a)
+ .25 P(S4b)

$$
\begin{aligned}
1.0 = \; & .25\,P(S1) \\
& + (.20)(.05)\,P(S1) + (.20)(.10)(.60)\,P(S1) \\
& + (.20)(.10)\,P(S1) + (.20)(.90)(.60)\,P(S1) \\
& + (.75)(.80)(.05)\,P(S1) + (.75)(.80)(.10)(.60)\,P(S1) \\
& + (.75)(.80)(.10)\,P(S1) + (.75)(.80)(.90)(.60)\,P(S1) \\
& + (.25)(.80)(.05)\,P(S1) + (.25)(.80)(.10)(.60)\,P(S1) \\
& + (.25)(.80)(.10)\,P(S1) + (.25)(.80)(.90)(.60)\,P(S1) \\
& + .60\,P(S1) \\
& + .10\,P(S1) + (.90)(.60)\,P(S1) \\
& + .05\,P(S1) + (.10)(.60)\,P(S1) \\
& + .10\,P(S1) + (.90)(.60)\,P(S1) \\
& + (.80)(.05)\,P(S1) + (.80)(.10)(.60)\,P(S1) \\
& + (.80)(.10)\,P(S1) + (.80)(.90)(.60)\,P(S1) \\
& + (.25)(.80)(.05)\,P(S1) + (.25)(.80)(.10)(.60)\,P(S1) \\
& + (.25)(.80)(.10)\,P(S1) + (.25)(.80)(.90)(.60)\,P(S1)
\end{aligned}
$$

$$1.0 = 3.74\,P(S1)$$

$$P(S1) = .26738$$

The shared-virtual-memory write availability of the new system with a variable delay, is lower than that of the system depicted in the previous section. This is what we would expect, the shortest delay in the variable delay system is equal to the fixed delay in the previous system. The results are more meaningful when they are compared with the results of other like systems. It should again be emphasized that small, or any, changes in parameters pertaining to gathered statistics can be incorporated into the model very easily, a claim that trace-driven simulation can not supply. It clearly shows the superiority of the proposed approach when dealing with a given system and for different memory access patterns.

5.6 The Page Manager Function

The function of the page manager is to keep track of which processor currently owns the page and which processors have copies of the page, it is this function the varies most significantly among different shared-virtual-memory protocols. In the example of a

shared-virtual-memory algorithm given in the previous chapter, it was assumed that faulting processors would be able to easily find the page manager. One method used to ensure that faulting processors can quickly find the page manager is to assign the page manager to a predetermined processor. This method is sometimes referred to as the centralized page management algorithm. The centralized method, while it is easy to implement, operates grossly inefficiently. The two main drawbacks to this centralized page management algorithm are due to the additional communication traffic centered around the page manager itself and the additional processing that must be performed by the page manager processor.

We first examine the issue of the additional communication, centered around the page manager. Every shared-virtual-memory page fault that occurs automatically generates a message which is sent to the page manager. This message is a request for a copy of the page. The page manager, in turn, must send a message to the owner of the particular page requested. This message to the owner is a request for a copy of the page to be sent to the faulting processor. If the original fault was a write fault, the page manager must also send invalidation messages to each processor which currently has a copy of this page. These additional messages in the network, which travel to and from the page manager, could create a communication traffic jam around and near the page manager processor. This traffic jam, caused by shared-virtual-memory communication messages, would degrade the performance of the shared-virtual-memory system altogether.

The second main drawback of a centralized page management system is the additional processing needed for the page manager function. If the processor, which ran the page manager, was not dedicated to the page manager function, its other tasks could be delayed, since every time a fault occurs within the shared-virtual-memory system, the page manager processor must be interrupted to handle the fault.

For the reasons described above and the fault critical nature of all centralized routing and page management approaches a more realistic page management algorithm is required. The page management protocol should have all or at least many processors sharing the page management functions. This would be called a distributed page manager. The actual page management task is distributed across multiple processing nodes in the system. A distributed page manager does not only balance the work performed by the page manager processors, but also utilize the communication network more efficiently. The traffic patterns in the network are no longer centered around a single node but hopefully are evenly distributed throughout the whole network.

Once it is determined to distribute the task of the page manager, the question is how? Which pages should be maintained by which processors? This question can be answered on a static or dynamic basis. Pages can be pre-assigned to processors for management, before the shared-virtual-memory system is started. This static assignment of pages to processors can be stored in a table on all processors. The task of finding the page manager now becomes simply a table look-up function. This is the assumption used in the proceeding model. This assumption implies that the page manager is quick and easy to find. It also implies that no single processor is burdened with the task of page management.

A more complex distributed manager is now analyzed. To improve the overall performance of the shared-virtual-memory system, the page manager function can be dynamically distributed. A dynamically distributed page manager would allow the manager function of individual pages to migrate from processor to processor. This can be viewed as a form of load balancing. One method could be to have the owner of a page keep track of all processors which currently have a copy of the page. When a new processor requires write access, it takes ownership from the previous owner, along with the list of

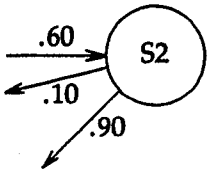
processors which currently have a copy of the page. The new owner then sends out invalidation messages to all processors which currently have a copy of the page. The task of finding the owner of the page now becomes a much more difficult task.

5.7 A Dynamic Distributed Page Manager

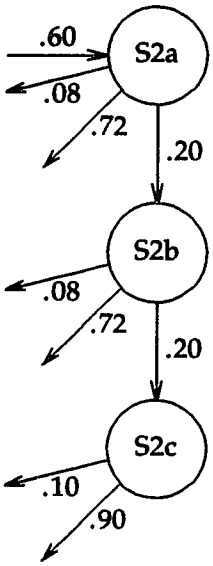
The performance of a dynamic distributed page manager algorithm is analyzed in this section. The shared-virtual-memory protocol and algorithm given in section 5.5 is augmented to contain a dynamic distributed page manager. Dynamic distributed page management protocols (algorithms) assume that the task of page management can migrate from processor to processor. In the newly presented and augmented algorithm, each processor keeps information pertaining to all shared-virtual-memory pages in the system. This information, that is kept in each processor, contains the most likely current page manager for the page. If the processor listed as page manager in the page table is not the real manager of the page, i.e., it relinquished management of the page to a new processor, messages are still sent to that processor which, in turn, passes them on to the new page manager. This process of passing the pages on eventually finds the correct page manager. When the correct page manager is found, all processors in the path update their tables to point to the new page manager. All subsequent requests to that page, made by any processor in that chain, automatically go to the correct manager until the management processor is changed again. The task of finding a page manager is similar to network routing and almost any network routing algorithm could be used to satisfy this task. Fore more information on network routing algorithms the reader should refer to either [Tane81] or [Davi79].

The performance model for a variable-time invalidation function initially presented in section 5.3 is now augmented to contain the dynamic distributed manager. To

illustrate the model development for a dynamic distributed page manager, some particular statistics are assigned to the task of finding the page manager, this choice of statistics is for illustrative purposes only, they do not imply anything else but an attempt to be specific. It is assumed that the page manager for 80% of all page faults, in which the faulting processor is not the page manager, is found in one unit of time. It also is further assumed that the page manager for 16% of all page faults, in which the faulting processor is not the page manager, is found in two units of time. The page manager for the remaining 4% of all page faults, in which the faulting processor is not the page manager, is found in three time units. Figure 5.6a shows the original page manager as given by state S2 of figure 5.5 and figure 5.6b shows the resulting states, i.e., states S2a, S2b, and S2c cloned from the original state S2 of figure 5.5.



Page Manager Before Cloning
Figure 5.6a

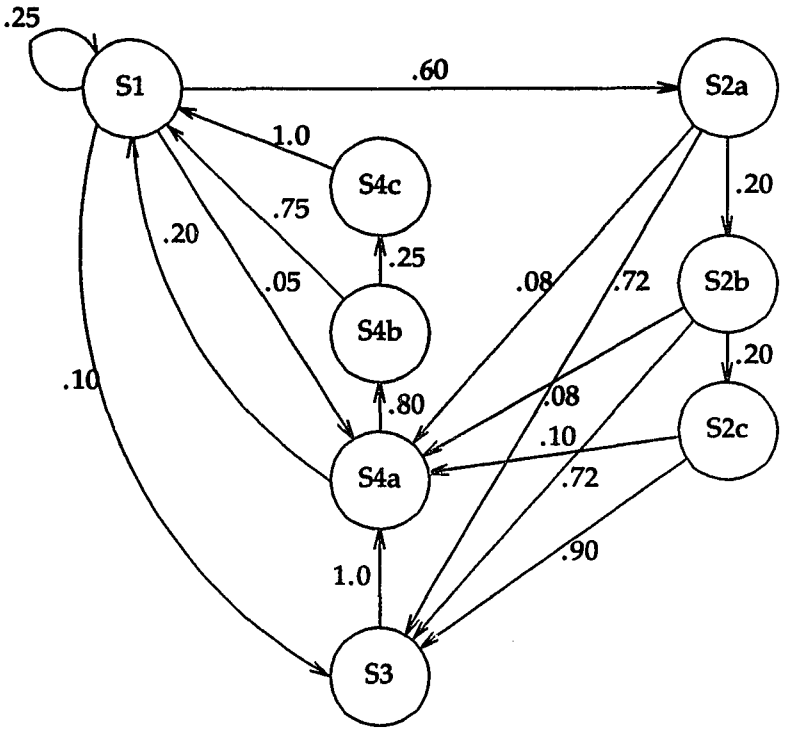


Cloned Page Manager
Figure 5.6b

The original arc entering state S2 in figure 5.6a enters state S2a, after cloning, in figure 5.6b. Since the clone is a variable time delay clone, all arcs leaving the original state also

leave each of the cloned states. If we look at state S2a in figure 5.6b, we see the original entry arc, the two exit arc plus one additional exit arc which leads to S2b. The magnitude of the exit arcs must be adjusted for the additional exit arc. Notice that the arc from S2a to S2b with a magnitude of .20, it is responsible for the fact that 20% of all references to the page manager are made in two or three time units, or that 80% of all references to the page manager are found in one time unit. Once we take away 20% of the output capacity of a state, as given by the new arc labeled .20, there is only 80% left. For this reason we multiply each of the original outgoing arcs, i.e., the two arcs of .10 and .90 respectively, by .80. After the multiplication, of the two original outgoing arcs by .80, they are now labeled as: .08 and .72. State S2b also has an additional output arc labeled .20, this arc represents the 4% of all page faults which require three units of time to find the page manager. Note that the 4% figure is derived by taking 20% of S2b which is only 20% of S2a, i.e., 20% of 20% which is 40% or .04. The outgoing arcs from state S2c are the same as the original outgoing arcs of that state before cloning, this is the last state in the cloned chain, and represents the three time delays of the worst case scenario.

Figure 5.7 below gives the complete state-transition diagram for the dynamic distributed page manager.



State-transition diagram of a dynamic distributed manager
Figure 5.7

The arcs in figure 5.7 are now explained:

S1 -> S1: The arc from S1 to S1, labeled .25, represents the 25% of all write instructions that are satisfied without a write fault. The page of memory which contains the desired location exists in the local processor's attached memory and is "read-write".

S1 -> S2a: The arc from S1 to S2a, labeled .60, represents the 60% of all write instructions that have encountered a write fault and the faulting processor is not the manager of the page in fault.

S1 -> S3: The arc from S1 to S3, labeled .10, represents 10% of all write instructions. These write instructions have encountered a write fault, and the faulting pro-

cessor is the manager but not owner of the page containing the desired location for writing.

S1 -> S4a: The arc from S1 to S4a, labeled .05, represents 5% of all write instructions. These write instructions have encountered a write fault and the faulting processor is both the manager and owner of the page containing the desired location for writing.

S2a -> S2b The arc from S2a to S2b, labeled .20, represents 20% of all write instructions in which the faulting processor is not the page manager These write instructions require more than one unit of time to find the page manager.

S2a -> S3 The arc from S2a to S3, labeled .72, represents 72% of write faults in which the faulting processor is not the manager of the page in fault. These 72% of all S1 -> S2a transitions are the write faults that occur when the page manager is not the owner, not the faulting processor, and found in one time unit.

S2a -> S4a The arc from S2a to S4a, labeled .08, represents the write faults which occur when the page manager is not the faulting processor, but the page manager is the owner and can be found in one unit of time.

S2b -> S2c The arc from S2b to S2c, labeled .20, represents 4% (20% of 20%) of all write instructions in which the faulting processor is not the page manager. These write instructions require more than two units of time to find the page manager.

S2b -> S3 The arc from S2b to S3, labeled .72, represents the write faults in which the faulting processor is not the manager page, the manager is not the owner, and the manager can be found in two units of time.

S2b -> S4a The arc from S2b to S4a, labeled .08, represents the write faults which occur when the page manager is not the faulting processor, but the page manager is the owner and can be found in two units of time.

S2c -> S3 The arc from S2c to S3, labeled .90, represents the write faults in which the faulting processor is not the manager page, the manager is not the owner, and the manager can be found in three units of time.

S2c -> S4a The arc from S2c to S4a, labeled .10, represents the write faults which occur when the page manager is not the faulting processor, but the page manager is the owner and can be found in three units of time.

S3 -> S4a The arc from S3 to S4a, labeled 1.0, represents 100% of all write instructions that encounter a fault, and the owner of the page is not the manager of the page or the faulting processor. This arc represents the situation that once the owner is located, all read-only pages in the system must be invalidated. States S4a, S4b, and S4c represents the invalidation of pages.

S4a -> S1: The arc from S4a to S1, labeled .20, represents 20% of all write instructions that have encountered a fault and have completed the invalidation in one of three possible time units.

S4a -> S4b: The arc from S4a to S4b, labeled .80, represents 80% of all write instructions that have encountered a fault and require more than one time unit for invalidation.

S4b -> S1: The arc from S4b to S1, labeled .75, represents 60% (75% of 80%) of all write instructions that have encountered a fault and have completed the invalidation in exactly two of three possible time units.

S4b -> S4c: The arc from S4b to S4c, labeled .25, represents 20% (25% of 80%) of all write instructions that have encountered a fault and require more than two time

units for invalidation.

S4c -> S1: The arc from S4c to S1, labeled 1.0, represents 100% of all write instructions that have encountered a fault and have completed the invalidation in exactly three of three possible time units.

The state-transition diagram given in figure 5.7 is a result of replacing state S2 in figure 5.5 with the variable delayed cloned states S2a, S2b, and S2c given in figure 5.6b and in accordance with anticipated statistical behaviors. The next step in analyzing the performance of the write operation of the shared-virtual-memory, with both invalidate and page manager functions set up as a variable delay, is to construct the state-transition matrix. Table 5.4 contains that matrix.

FROM	TO							
	S1	S2a	S2b	S2c	S3	S4a	S4b	S4c
S1	0.25	0.60	0.00	0.00	0.10	0.05	0.00	0.00
S2a	0.00	0.00	0.20	0.00	0.72	0.08	0.00	0.00
S2b	0.00	0.00	0.00	0.20	0.72	0.08	0.00	0.00
S2c	0.00	0.00	0.00	0.00	0.90	0.10	0.00	0.00
S3	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
S4a	0.20	0.00	0.00	0.00	0.00	0.00	0.80	0.00
S4b	0.75	0.00	0.00	0.00	0.00	0.00	0.00	0.25
S4c	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.4: State-transition matrix for a Dynamic Distributed Manager

After the state-transition matrix is constructed, the availability of the write operation into the shared-virtual-memory can now be computed with the same methodology as used in the previous section. The next-state probability equations of the write operation into a shared-virtual-memory with variable page invalidation and management delays are

given below:

$\hat{P}(S1)$ = 0.25 P(S1) + 0.20 P(S4a) + 0.75 P(S4b) + P(S4c)
$\hat{P}(S2a)$ = .60 P(S1)
$\hat{P}(S2b)$ = .20 P(S2a)
$\hat{P}(S2c)$ = .20 P(S2b)
$\hat{P}(S3)$ = .10 P(S1) + .72 P(S2a) + .72 P(S2b) + .90 P(S2c)
$\hat{P}(S4a)$ = .05 P(S1) + .08 P(S2a) + .08 P(S2b) + .10 P(S2c) + P(S3)
$\hat{P}(S4b)$ = .80 P(S4a)
$\hat{P}(S4c)$ = .25 P(S4b)

The write availability of the shared-virtual-memory system is computed using the matrix multiplication method presented in section 3.3. Below is the matrix representation of the probability of making a transition from state i to state j, where i is the state associated with the row and j is the state associated with the column, as given in Table 5.4.

$$P(i,j) = \begin{vmatrix} 0.250 & 0.600 & 0.000 & 0.000 & 0.100 & 0.050 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.200 & 0.000 & 0.720 & 0.080 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.200 & 0.720 & 0.080 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.000 & 0.900 & 0.100 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 1.000 & 0.000 & 0.000 \\ 0.200 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.800 & 0.000 \\ 0.750 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.250 \\ 1.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \end{vmatrix}$$

The system stabilizes to within 5 significant decimal places after 68 cycles, i.e., the difference between $P(i,j)^{68}$ and $P(i,j)^{69}$ is less than 0.000005 for all i and j.

$$P(i,j)^{68} = \begin{vmatrix} .25747 & .15448 & .03090 & .00618 & .16478 & .19310 & .15448 & .03862 \\ .25747 & .15448 & .03090 & .00618 & .16478 & .19310 & .15448 & .03862 \\ .25747 & .15448 & .03090 & .00618 & .16478 & .19310 & .15448 & .03862 \\ .25747 & .15448 & .03090 & .00618 & .16478 & .19310 & .15448 & .03862 \\ .25747 & .15448 & .03090 & .00618 & .16478 & .19310 & .15448 & .03862 \\ .25747 & .15448 & .03090 & .00618 & .16478 & .19310 & .15448 & .03862 \\ .25747 & .15448 & .03090 & .00618 & .16478 & .19310 & .15448 & .03862 \\ .25747 & .15448 & .03090 & .00618 & .16478 & .19310 & .15448 & .03862 \end{vmatrix}$$

As shown above, the write availability of the shared-virtual-memory system is .25747. The matrix multiplication was solved by using a program written in the "C" programing language [Kern78]. The matrix multiplication "C" program specifically

developed for this case is presented in appendix A.

5.8 A Modified Dynamic Distributed Page Manager

The dynamic distributed page manager, as presented in the previous section, is now modified to function more efficiently. The task of the page manager is moved to the owner. It now becomes the responsibility of the page owner to keep a record of every processor that has a copy of the page. With this new owner-manager protocol the owner of a page is the processor which is responsible for sending out the page invalidation messages. When a processor encounters a write fault, that processor must find the owner and request ownership. Once ownership is granted, the processor is given the page, along with the list of processors which currently have a copy of the page. It is now the task of the new page owner to send out invalidation messages to all processors which currently have a copy of the page. Once all copies of the requested page have been invalidated, the new owner can set the access of that page to read-write. At this point the page can be written to, and memory coherence is maintained. To better understand this new modified dynamic distributed page manager algorithm some particular statistics are assumed for illustrative purposes only. The first shared-virtual-memory write algorithm, from section 5.3, is modified. The same statistics are used to enable the results of both models to be compared.

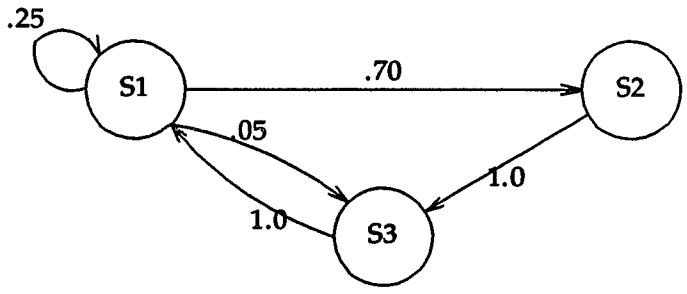
Assume, as in section 5.3, that 25% of all write requests are satisfied without a write fault. The remaining 75% of all write requests thus causes a write fault to occur. When a write fault occurs it can be classified into one of two different categories which are described below:

(1) The faulting processor is not the owner of the page. In this first category, a request for write access to a page would be made from the faulting processor to

the owner of the page. For this example it is assumed that this scenario is the case for 70% of all write instructions.

(2) The faulting processor is the owner of the page. This second category implies that the faulting processor currently has the page as read-only. Since the processor is the owner of the page, it must have a copy of the page, and this copy must be read-only otherwise a write fault would not have occurred. In this case all other read-only copies of the page must be invalidated before this page can be changed to read-write. It is assumed that this occurs for 5% of all write faults.

Figure 5.8 gives the state-transition diagram for the modified dynamic distributed page manager algorithm.



Modified shared-virtual-memory write state diagram
Figure 5.8

The arcs in figure 5.8 are now explained as below:

S1 -> S1: The arc from S1 to S1 labeled .25 represents the 25% of all write instructions that are satisfied without a write fault. The page of memory which contains the desired location exists in the local processor's attached memory and is read-write.

S1 -> S2: The arc from S1 to S2, labeled .70, represents the 70% of all write instructions that have encountered a write fault and the faulting processor is not the

owner of the page in fault.

S1 -> S3: The arc from S1 to S3, labeled .05, represents 5% of all write instructions. These write instructions have encountered a write fault and the faulting processor is the owner of the page which contains the desired location for writing. When this fault occurs, the page in the attached memory of the local processor must be marked read-only and all copies of the page must be invalidated by this transition as well.

S2 -> S3: The arc from S2 to S3, labeled 1.0, represents 100% of all write faults in which the faulting processor is not the owner of the page in fault. This transition represents the need for copies of the page to be invalidated.

S3 -> S1: The arc from S3 to S1, labeled 1.0, represents 100% of all write instructions that have encountered a fault and have completed the invalidation of all other pages that contained the desired write location in the system. From this point in the protocol, we always return to the faulting processor to complete the write operation.

After constructing the state-transition diagram (see figure 5.8) the next step in analyzing the write operation into the modified shared-virtual-memory system is to build a state-transition matrix. This matrix is constructed in the same manner as seen earlier in the shared-virtual-memory write model of section 5.3 and is given below in Table 5.5.

FROM	TO		
	S1	S2	S3
S1	0.25	0.70	0.05
S2	0.00	0.00	1.00
S3	1.00	0.00	0.00

Table 5.5: Modified shared-virtual-memory write state-transition matrix.

After the state-transition matrix is constructed, the availability of a write operation into the modified shared-virtual-memory system is now computed, using the same methodology as before. The next-state probability equations of the modified shared-virtual-memory write are given below:

$\hat{P}(S1)$ = .25 P(S1) + P(S3)
$\hat{P}(S2)$ = .70 P(S1)
$\hat{P}(S3)$ = .05 P(S1) + P(S2)

The availability of a write operation into the modified shared-virtual-memory system is computed to be .40816, as seen from the derivation that follows.

$\hat{P}(S1) + \hat{P}(S2) + \hat{P}(S3)$ = .25 P(S1) + P(S3)
+ .70 P(S1)
+ .05 P(S1) + P(S2)

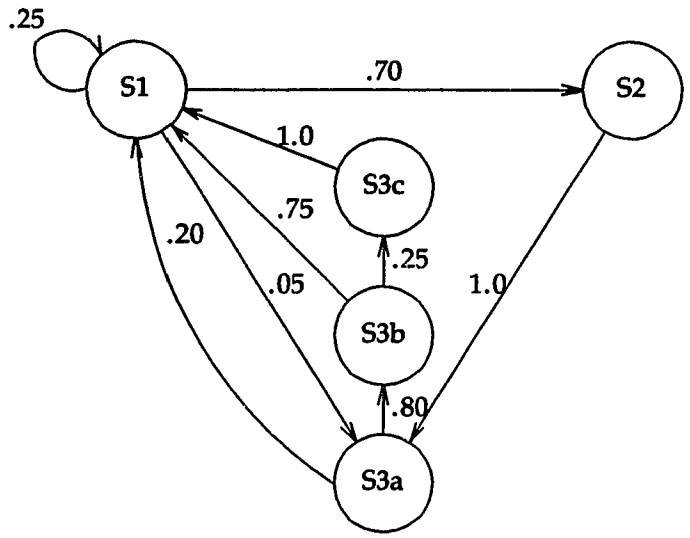
1.0 = .25 P(S1) + .05 P(S1) + .70 P(S1)
+ .70 P(S1)
+ .05 P(S1) + .70 P(S1)

1.0 = 2.45 P(S1)

P(S1) = .40816

The modified model, described in this section, has a better write availability rating than the model in section 5.3. For comparison purposes, it should be noted that before the modification, the model had an availability rating of .33445.

The modified system is now examined with a variable time invalidation function. The availability of this system is compared to the availability of the system described in section 5.5. In order for the comparison of the two systems to be valid, the invalidation state must be cloned with the same assumptions and in accordance with the procedure developed in chapter 3 section 3.5. These assumptions are that 20% of all page invalidation operations take place in one unit of time, 60% of all page invalidation operations take place in two time units, and the remaining 20% of all page invalidation operations take place in three time units of time. Figure 5.9 shows the modified distributed manager algorithm with a variable time invalidation function. Notice that in figure 5.9, state S3 from the previous model (see figure 5.8) has been cloned into three states, i.e., states S3a, S3b, and S3c, as required.



State-transition diagram of a modified variable invalidate write.
Figure 5.9

The arcs in figure 5.9 are now explained:

S1 -> S1: The arc from S1 to S1 labeled .25 represents the 25% of all write instructions that are satisfied without a write fault. The page of memory which contains

the desired location exists in the local processor's attached memory and is read-write.

S1 -> S2: The arc from S1 to S2, labeled .70, represents the 70% of all write instructions that have encountered a write fault and the faulting processor is not the owner of the page in fault.

S1 -> S3a: The arc from S1 to S3a, labeled .05, represents 5% of all write instructions. These write instructions have encountered a write fault and the faulting processor is the owner of the page which contains the desired location for writing. The page must be marked read-only for this fault to occur. This transition represents the need for all copies of the page to be invalidated, as dictated by the protocol.

S2 -> S3a: The arc from S2 to S3a, labeled 1.0, represents 100% of all write faults in which the faulting processor is not the owner of the page in fault. The transition represented by this arc is due to the need for all copies of this page to be invalidated.

S3a -> S1: The arc from S3a to S1, labeled .20, represents 20% of all write faults. The invalidation associated with these write faults are accomplished in one time unit.

S3a -> S3b: The arc from S3a to S3b, labeled .80, represents the 80% of all write faults that require more than one time unit for invalidation.

S3b -> S1: The arc from S3b to S1, labeled .75, represents 60% (75% of 80% is indeed 60%) of all write faults. The invalidation associated with these write faults are accomplished in two time units.

S3b -> S3c: The arc from S3b to S3c, labeled .25, represents the 20% (25% of 80%, i.e., 20%) of all write faults that require more than two time units for invalidation.

S3c -> S1: The arc from S3c to S1, labeled 1.0, represents 20% (1.0 * .25 * .80) of all write faults. The invalidation associated with these write faults are accomplished in three time units.

After constructing the state-transition diagram (see figure 5.9) the next step in analyzing the modified shared-virtual-memory write instruction, with a variable invalidation delay, is to build a state transition matrix. This matrix is given in Table 5.6.

FROM	TO				
	S1	S2	S3a	S3b	S3c
S1	0.25	0.70	0.05	0.00	0.00
S2	0.00	0.00	1.00	0.00	0.00
S3a	0.20	0.00	0.00	0.80	0.00
S3b	0.75	0.00	0.00	0.00	0.25
S3c	1.00	0.00	0.00	0.00	0.00

Table 5.6: Modified Shared-virtual-memory variable invalidate matrix.

After the state-transition matrix is constructed, the shared-virtual-memory write availability can now computed. The next-state probability equations of the modified shared-virtual-memory write with variable page invalidation delay, are given below:

$\hat{P}(S1)$ = 0.25 P(S1) + 0.20 P(S3a) + 0.75 P(S3b) + P(S3c)
$\hat{P}(S2)$ = .70 P(S1)
$\hat{P}(S3a)$ = .05 P(S1) + P(S2)
$\hat{P}(S3b)$ = .80 P(S3a)
$\hat{P}(S3c)$ = .25 P(S3b)

The write availability of the shared-virtual-memory write system is computed by using the matrix multiplication method presented above and in section 3.3. Below is the matrix representation of the probability of making a transition from state i to state j,

where i is the state associated with the row and j is the state associated with the column, as given in Table 5.6.

$$P(i,j) = \begin{vmatrix} 0.250 & 0.700 & 0.050 & 0.000 & 0.000 \\ 0.000 & 0.000 & 1.000 & 0.000 & 0.000 \\ 0.200 & 0.000 & 0.000 & 0.800 & 0.000 \\ 0.750 & 0.000 & 0.000 & 0.000 & 0.250 \\ 1.000 & 0.000 & 0.000 & 0.000 & 0.000 \end{vmatrix}$$

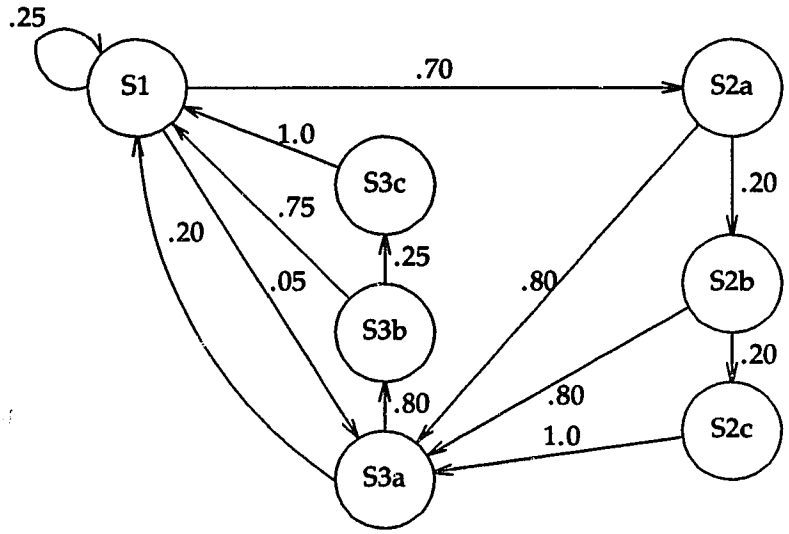
The system stabilizes to within 5 significant decimal places after 58 cycles.

$$P(i,j)^{58} = \begin{vmatrix} .31250 & .21875 & .23438 & .18750 & .04687 \\ .31250 & .21875 & .23438 & .18750 & .04687 \\ .31250 & .21875 & .23438 & .18750 & .04687 \\ .31250 & .21875 & .23438 & .18750 & .04687 \\ .31250 & .21875 & .23438 & .18750 & .04687 \end{vmatrix}$$

As shown above, the write availability of the modified shared-virtual-memory system is .31250. The rating of the modified system is better than that of the same system before modification. For comparison in section 5.5 the system, without modification, was computed to have a availability rating of .26738.

The modified system is now examined in its final form, with a variable time delay associated with finding the owner of a page. Up until now the modified system has be examined assuming that the owner of a page can be found in one unit of time. This is not always the case, and a more realistic assumption would be a variable time delay. To illustrate the impact of a variable time delay associated with finding the owner of a page, some statistical assumptions must be made. The assumptions, associated with the time involved in finding the owner of a page, are the same as the assumptions made for finding the manager of a page, in section 5.7. These statistical assumptions are kept the same so we can compare the results of the modified algorithm to that of the algorithm before modification.

It is assumed that the owner of a page can be found in one unit of time for 80% of all page faults, in which the faulting processor is not the owner of the page. It is also assumed that the page owner can be found in two units of time for 16% of all page faults, in which the faulting processor is not the page owner. The page owner is found in three units of time for the remaining 4% of all page faults, in which the faulting processor is not the page owner. Figure 5.10 shows the complete state diagram for this modified system.



State-transition diagram of a modified dynamic distributed manager
Figure 5.10

The arcs in figure 5.10 are now explained:

S1 -> S1: The arc from S1 to S1, labeled .25, represents the 25% of all write instructions that are satisfied without a write fault. The page of memory which contains the desired location exists in the local processor's attached memory and is "read-write".

S1 -> S2a: The arc from S1 to S2a, labeled .70, represents the 70% of all write instructions that have encountered a write fault and the faulting processor is not the

owner of the page in fault.

S1 -> S3a: The arc from S1 to S3a, labeled .05, represents 5% of all write instructions. These write instructions have encountered a write fault and the faulting processor is the owner of the page containing the desired memory location.

S2a -> S2b The arc from S2a to S2b, labeled .20, represents 20% of all write instructions in which the faulting processor is not the page owner. These write instructions require more than one unit of time to find the owner of the page.

S2a -> S3a The arc from S2a to S3a, labeled .80, represents 80% of write faults in which the faulting processor is not the owner of the page. In these write faults the owner of the page can be found in exactly one unit of time.

S2b -> S2c The arc from S2b to S2c, labeled .20, represents 4% (20% of 20%) of all write instructions in which the faulting processor is not the page owner. These write instructions require more than two units of time to find the owner of the page.

S2b -> S3a The arc from S2b to S3a, labeled .80, represents 16% (80% of 20%) of write faults in which the faulting processor is not the owner of the page. In these write faults the owner of the page can be found in exactly two units of time.

S2c -> S3a The arc from S2c to S3a, labeled 1.0, represents 16% (100% of 20% of 20%) of write faults in which the faulting processor is not the owner of the page. In these write faults the owner of the page can be found in exactly three units of time.

S3a -> S1: The arc from S3a to S1, labeled .20, represents 20% of all write instructions that have encountered a fault and have just completed the invalidation in one of three possible time units.

S3a -> S3b: The arc from S3a to S3b, labeled .80, represents 80% of all write instructions that have encountered a fault and require more than one time unit for invalidation.

S3b -> S1: The arc from S3b to S1, labeled .75, represents 60% (75% of 80%) of all write instructions that have encountered a fault and have just completed the invalidation in exactly two of three possible time units.

S3b -> S3c: The arc from S3b to S3c, labeled .25, represents 20% (25% of 80%) of all write instructions that have encountered a fault and require more than two time units for invalidation.

S3c -> S1: The arc from S3c to S1, labeled 1.0, represents 100% of all write instructions that have encountered a fault and have just completed the invalidation in exactly three of three possible time units.

The state-transition diagram given in figure 5.10 is a result of replacing state S2, in figure 5.9, with the variable delayed cloned states S2a, S2b, and S2c according to the cloning procedure developed in section 3.5. It should be noted that this variable delay clone is used to represent the variable delay associated with finding the owner of the required page. The next step in analyzing the performance of the modified shared-virtual-memory write, with both invalidate and page owner functions modeled and set up as a variable delay, is to construct the state-transition matrix. Table 5.7 contains that matrix.

FROM	TO						
	S1	S2a	S2b	S2c	S3a	S3b	S3c
S1	0.25	0.70	0.00	0.00	0.05	0.00	0.00
S2a	0.00	0.00	0.20	0.00	0.80	0.00	0.00
S2b	0.00	0.00	0.00	0.20	0.80	0.00	0.00
S2c	0.00	0.00	0.00	0.00	1.00	0.00	0.00
S3a	0.20	0.00	0.00	0.00	0.00	0.80	0.00
S3b	0.75	0.00	0.00	0.00	0.00	0.00	0.25
S3c	1.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.7: Modified Dynamic Distributed Manager State-transition matrix

After the state-transition matrix is constructed, the write availability of modified shared-virtual-memory system can now be computed. The next-state probability equations of the modified shared-virtual-memory write with variable page invalidation and management delays are given below:

$\hat{P}(S1)$ = 0.25 P(S1) + 0.20 P(S3a) + 0.75 P(S3b) + P(S3c)
$\hat{P}(S2a)$ = .70 P(S1)
$\hat{P}(S2b)$ = .20 P(S2a)
$\hat{P}(S2c)$ = .20 P(S2b)
$\hat{P}(S3a)$ = .05 P(S1) + .80 P(S2a) + .80 P(S2b) + P(S2c)
$\hat{P}(S3b)$ = .80 P(S3a)
$\hat{P}(S3c)$ = .25 P(S3b)

The write availability of the modified shared-virtual-memory system is computed using the matrix multiplication method. For convince, Appendix A lists a "C" program specifically developed for these cases. Below is the matrix representation of the probability of making a transition from state i to state j, where i is the state associated with the row and j is the state associated with the column, as given in Table 5.7.

$$P(i,j) = \begin{vmatrix} 0.250 & 0.700 & 0.000 & 0.000 & 0.050 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.200 & 0.000 & 0.800 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.200 & 0.800 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.000 & 1.000 & 0.000 & 0.000 \\ 0.200 & 0.000 & 0.000 & 0.000 & 0.000 & 0.800 & 0.000 \\ 0.750 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.250 \\ 1.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \end{vmatrix}$$

The system stabilizes to within 5 significant decimal places after 54 cycles.

$$P(i,j)^{54} = \begin{vmatrix} .29691 & .20784 & .04157 & .00831 & .22268 & .17815 & .04454 \\ .29691 & .20784 & .04157 & .00831 & .22268 & .17815 & .04454 \\ .29691 & .20784 & .04157 & .00831 & .22268 & .17815 & .04454 \\ .29691 & .20784 & .04157 & .00831 & .22268 & .17815 & .04454 \\ .29691 & .20784 & .04157 & .00831 & .22268 & .17815 & .04454 \\ .29691 & .20784 & .04157 & .00831 & .22268 & .17815 & .04454 \\ .29691 & .20784 & .04157 & .00831 & .22268 & .17815 & .04454 \end{vmatrix}$$

As shown above, the write availability of the modified shared-virtual-memory system is .29691. The rating of the modified system is clearly better than that of the system before modification. For illustration purposes, in section 5.7 the original system, i.e., without modification, was computed to have a availability rating of .25747.

5.9 Summary

This chapter develops a framework for analyzing the performance of shared-virtual-memory systems. First, a simple read operation was examined, followed by the more interesting write operation. The write operation was first analyzed in its simplest form, followed by the more complex case of variable time delays. The variable time delay was used to analyze the page invalidation protocol function and a distributed page manager algorithm. The shared-virtual-memory algorithm was then enhanced in order to perform more efficiently. This is accomplished by assigning the page managers task to the owner of the page. This assignment would eliminate the page manager and thus save at least

one page request in most page faults. The complete analyses was then performed on the improved algorithm. The memory access statistical patterns assumed in the analyses were the same as those used with the prior algorithm, this facilitated the realistic comparison of the results between the old and the new algorithm. In all three variations of the protocol model, the analysis performed indicate that the new algorithm performed substantially better.

Chapter 6

Conclusions

This dissertation presented an analytical framework for analyzing the performance of the memory hierarchy. A framework was developed and then applied to a traditional memory hierarchy which consisted of cache and main memory. An example of a shared-virtual-memory system was given and the performance model was then applied to this complex hierarchical memory system. Memory coherency protocols were also presented and their performance was examined and compared.

6.1. Summary Of Results

The dissertation presents an innovative analytical approach to the evaluation of hierarchical memory systems. A performance model based, on Markov chains, was developed and presented in chapter 3. This performance model was first applied to a simple three level memory hierarchy, which consisted of cache and main memory. The notion of system availability was then defined as a measure of the memory hierarchy performance. The performance evaluation using this model was then further enhanced to model non-equal fixed time delays for different levels of the memory hierarchy. This non-equal fixed time delay modeling was accomplished through the use of state cloning

techniques which were originally developed and presented in chapter 3. Some additional design improvements to the memory hierarchy, such as better page replacement algorithm or a larger cache, that would affect the performance of the memory system were applied to the new memory hierarchy, and the resulting improvement in the memory system performance was analyzed. The performance model was then enhanced to handle non-equal fixed time delays. The results of all these improved memory systems were then compared with that of the prior memory system model. A more complex memory system, with one level of the memory hierarchy incurring a variable time delay, was then presented, and new state cloning techniques were then developed, which enabled the performance model to analyze the variable time delay. The variable time delay hierarchical memory system was then analyzed and the results were compared with that of the non-equal fixed time delay system. Variable time delay capabilities are obviously critical when analyzing cache or virtual memory, because the amount of time required for a memory access is definitely different, depending upon whether or not the desired location is currently present in a particular level of the memory hierarchy.

The non-equal fixed delay and variable delay enhancements made to the performance model in chapter 3 are necessary for the modeling and analysis of the memory coherency protocols. As explained in chapter 4, memory coherency protocols are necessary when implementing shared-virtual-memory. Shared-virtual-memory transforms the memory access behavior of a loosely-coupled parallel processing system into that of a tightly-coupled system with shared-global-memory. This transformation gives software developers an alternative form of inter-processor communication which has the advantage of being transparent to the programer. Data from one processor can be written into shared-virtual-memory that another processor can read. Without the shared-virtual-memory system, the programer would have to explicitly send the data in a communication message. Efficient shared-virtual-memory implementations allow multiple copies of

a single memory location to exist, i.e., one in each processor. In order to maintain memory coherency, a write operation can only be performed on a single existence of a memory location, i.e., no other copies may exist. Shared-virtual-memory algorithms through the use of memory coherency protocols are used to invalidate all other copies of a memory locations before write operations take place in one copy only. Chapter 4 explains in detail the issues associated with memory coherency and gives an example of a shared-virtual-memory algorithm.

The shared-virtual-memory system, described in chapter 4, was then analyzed in chapter 5 using the same performance model which was developed in chapter 3. A read operation on a shared-virtual-memory was analyzed first and the read availability of the system was then computed. The shared-virtual-memory write operation, which is more complex due to the page invalidation protocol requirements, was subsequently analyzed using a fixed time delay page invalidation function. A more realistic, variable time delay, assumption was then made for the page invalidation protocol function and a new model was then constructed using the state cloning techniques developed in chapter 3. The results from the variable time delay page invalidation protocol system were then compared with the results of the fixed time delay system.

Dynamically distributed page management algorithms, as the name eludes to, dynamically distributes the task of page management to different processors within the domain of the shared-virtual-memory system. The shared-virtual-memory system was then enhanced in such a way as to contain a dynamically distributed page management algorithm and was then modeled with a variable time delay page management protocol to represent the new algorithm used. The write availability for the dynamic distributed shared-virtual-memory system was then computed using the alternative approach of the matrix multiplication method. A matrix multiplication routine specifically developed for

this thesis and written in the "C" programing language is also given (Appendix A) to help in solving the system availability of large complex systems.

The dynamic distributed page manager was then further modified in order to allow it to function more efficiently. This modification entailed merging the tasks of the page manager into that of the page owner. The responsibility of keeping a record of every processor that has a copy of the page becomes that of the page owner and thus allows the elimination of the page manager per se. The owner of a page is also responsible for sending out page invalidation messages. This new owner-manager protocol was then modeled to ascertain if indeed it is more efficient than the original dynamic distributed page manager approach. Variable time delays, state cloning techniques, and the alternative matrix multiplication method were all used to determine the write availability of this new shared-virtual-memory system. The results were then compared to that of the prior system and indeed, as expected, the modified dynamic distributed page management shared-virtual-memory system clearly performed better.

6.2. Related Work

Much work in the performance evaluation of hierarchical memory systems is based upon trace-driven simulation. In trace-driven simulation, traces of a computer program's memory access are collected. The memory access pattern's affect on different organizations of the memory hierarchy is then simulated. The affect of different cache coherency protocols are also determined through simulation. Simulation is very costly in terms of the amount of CPU time consumed and disk space needed. In [Wan89a] a method of reducing traces is given to alleviate the required disk space requirement and decrease CPU computation time needed when employing simulation. Parallel algorithms that help reduce the time needed for a trace driven simulation have also been developed [Lin89]

and [Lin88].

A model for cost measurement of cache coherency protocols in loosely-coupled multiprocessors is given in [Li86b]. In that model, the cost of a page fault is given in terms of the cost of sending a message, receiving a message and invalidating a page.

Several different organizations of multilevel cache hierarchies are presented with performance evaluation in [Wang89b], and the performance evaluation method used is that of trace-driven simulation. In the future research section of [Wang89b] it is stated that there are several problems related to trace-driven simulation, such as the amount of time and storage space required. One minute of execution time from a 150 MIP multiprocessor would generate a trace of more than one billion instructions. However, aside from the size of the trace, collecting a meaningful trace is also a problem. [Wab89b] also states that the validity of this type of simulation has not yet been fully studied in the sense that simulation results of one system does not necessarily depict the behavior of another, different system. Analytical models, such as the one described in this research, thus constitute a viable alternative technique to the trace-driven simulation used so far.

6.3. Future Research

The performance of the hierarchical memory models given in this dissertation were measured with the steady-state probability of a constructed performance model. A supplement to the steady-state analysis could be transient analysis of the same constructed model. For example, the very first memory system analyzed shown in figure 3.1 stabilized after 22 transitions. The time that a system takes to arrive at the steady-state is also a measure of the performance of the system.

The performance model developed in this dissertation, although exclusively applied to hierarchical memory systems, can also be used on a wide variety of applications. One

such application, that of communication network protocol efficiency, comes to mind immediately. The methods used in applying the performance model to the dynamic distributed shared-virtual-memory system, of figure 5.10 in chapter 5, could also be used in applying the model to a communication network routing algorithm.

6.4. Final Thoughts

This dissertation presents a novel approach that can act as an alternative or complement to simulation. This approach enjoys a definite improvement over simulation by allowing one to use analytical tools that pure simulation does not offer. Even though the discussion and examples pertained to memory systems, the model is by no means constrained to this application. This model can be used on a broad spectrum of similar problems with ease and similar success. A theoretical approach is always superior to pure simulation, not only because it yields a closed-form solution, but it also because it provides an alternative avenue of solution and a different prospective to the same problem.

Appendix A

C Program For Matrix Multiplication

```
 1 #include    <stdio.h>
 2
 3 /* Define Maximum Matrix Size */
 4 #define     MTX_MAX 10
 5
 6 double      mtx[MTX_MAX][MTX_MAX];          /* Initial Matrix   */
 7 double      mtx1[MTX_MAX][MTX_MAX];         /* Final Matrix     */
 8 double      tmp[MTX_MAX][MTX_MAX];          /* Temporary Matrix */
 9 int         m_size;                         /* Size of Matrix   */
10
11 /*
12  * This program will prompt the user for the matrix size,
13  * original matrix for multiplication, and the power to
14  * which the matrix should be raised
15  * The output will be the initial matrix raised to the
16  * given power
17  */
18
19 main()
20 {
21     int     i, power, atoi();
22     char    buf[80];
23
24     /*
25      * Get original matrix size
26      */
27     printf("Enter Matrix Size: ");
28     gets(buf);
29     m_size = atoi(buf);
30     if ( m_size > MTX_MAX ) {
31             printf("Error matrix can be no larger than %d\n",MTX_MAX);
32             exit(0);
33     }
34
35     /*
36      * Get the elements of the original matrix
```

```
37      */
38      for ( i=0; i < m_size; i++ ) {
39              printf("enter row %d: ",i);
40              gets(buf);
41              sscanf(buf,"%lf %lf %lf %lf %lf %lf %lf %lf %lf %lf",
42                      &mtx[i][0],&mtx[i][1],&mtx[i][2],
43                      &mtx[i][3],&mtx[i][4],&mtx[i][5],
44                      &mtx[i][6],&mtx[i][7],&mtx[i][8], &mtx[i][9]);
45      }
46
47      /*
48       * Print the input matrix
49       */
50      printf("input matrix\n");
51      pr_mtx(mtx);
52
53      /*
54       * Prompt the user for power to raise matrix to,
55       * raise matrix to power through multiplication
56       * print final answer and prompt for new power
57       */
58      for ( ;; ) {
59              printf("Enter power to raise to: ");
60              gets(buf);
61              power = atoi(buf);
62              if ( power == 0 ) break;
63              printf("Matrix to the %d power\n",power);
64              cp_mtx(mtx,mtx1);
65              for ( i=1; i<power; i++ ) {
66                      cp_mtx(mtx1,tmp);
67                      mtx_mult(tmp,mtx,mtx1);
68              }
69              pr_mtx(mtx1);
70      }
71  }
72
73  /*
74   * cp_mtx() : Copy from matrix to to matrix
75   */
76  cp_mtx(from,to)
77  double      from[MTX_MAX][MTX_MAX];
78  double      to[MTX_MAX][MTX_MAX];
79  {
80      int     i,j;
81      for ( i=0; i < m_size; i++ )
82              for ( j=0; j < m_size; j++ )
83                      to[i][j] = from[i][j];
84  }
85
86  /*
87   * mtx_mult(): Multiply matrix m1 by m2 and place the
```

```
 * result in matrix m3
 */
mtx_mult(m1,m2,m3)
double      m1[MTX_MAX][MTX_MAX];
double      m2[MTX_MAX][MTX_MAX];
double      m3[MTX_MAX][MTX_MAX];
{
    int     i,j,k;
    double  tmp;

    for ( i=0; i < m_size; i++ ) {
            for ( j=0; j < m_size; j++ ) {
                    m3[i][j] = 0.0;
                    for ( k=0; k < m_size; k++ ) {
                            m3[i][j] += m1[i][k] * m2[k][j];
                    }
            }
    }
    return 0;
}

/*
 * pr_mtx(): print matrix xmtx to the standard output
 */
pr_mtx(xmtx)
double      xmtx[MTX_MAX][MTX_MAX];
{
    int     i,j;

    for ( i=0; i < m_size; i++ ) {
            for ( j=0; j < m_size; j++ ) {
                    printf("%7.5lf  ",xmtx[i][j]);
            }
            printf("\n");
    }
}
```

Bibliography

[Arla88] Arlauskas, R., iPSC/2 System: A Second Generation Hypercube., *Proceedings of the Third Hypercube Conference.*, ACM, 1988.

[Baer89] Baer, J.-L., and Wang, W.-H. Multilevel Cache Hierarchies: Organizations, Protocols, and Performance, *Journal of Parallel and Distributed Computing 6*, pages 451-476, 1989

[Bril87] Bril, R. J., An Implementation Independent Approach to Cache Memories, *ACM Computer Architecture News*, Vol. 15, No 3, pages 17-24, June 1987

[Davi79] Davies, D.W., Barber,D.L.A, Price, W.L., Solomonides, C.M., *Computer Networks and Their Protocols*, Wiley, New York, 1979.

[Fell68] Feller, W., *An Introduction to Probability Theory and Its Applications*, 3rd Edition, Vol. 1, Wiley, New York, 1968.

[Flyn66] Flynn, M.J., Very High Speed Computing Systems, *Proceedings of IEEE*, pages 1901-1909, December 1966.

[Inte90] Intel Corp., *i860 64-Bit Microprocessor Programmer's Reference Manual*, 1990

[Gels89] Gelsinger, P. P., Gargini, P. A., Parker, P. H., Yu, A.Y.C., Microprocessors circa 2000, *IEEE Spectrum*, pages 43-47, October 1989

[Grov90] Groves, R.D., Oehler, R., RISC System/6000 Processor Architecture, *IBM RISC System/6000 Technology*, SA23-2619, IBM Corporation, pages 16-23, 1990

[Hard90] Hardell, R. H. Jr., Hicks, D. A., Howell, L. C. Jr., Maule, R. M., Montoye, R., Tuttle, D. P., Data Cache and Storage Control Units, *IBM RISC System/6000 Technology*, SA23-2619, IBM Corporation, pages 44-50, 1990

[Hwan84] Hwang, Kai., Briggs, Faye A., *Computer Architecture And Parallel Processing*, McGraw-Hill, New York, 1984

[Kern78] Kernighan, B.W., Ritchie, D.M., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978

[Klei75] Kleinrock, L., *Queueing Systems Volume 1: Theory*, Wiley, New York, 1975.

[Kohn89] Kohn, L., Fu, S.W., A 1,000,000 Transistor Microprocessor, *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, New York, NY, February 15-17, 1989.

[Li86a] Li, K., Hudak, P., Memory Coherence in Shared Virtual Memory Systems, *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 229-239, August 1986

[Li86b] Li, K., *Shared Virtual Memory On Loosely Coupled Multiprocessors*, PhD thesis, Yale University, Department Of Computer Science., September, 1986

[Lin88] Lin, Y. -B., Baer, J. -L., Lazowska, E., *Tailoring a Parallel Trace-Driven Simulation Technique to Specific Multiprocessor Cache Coherence Protocols*, University Of Washington, Department Of Computer Science. Technical Report 88-03-02, March 1988

[Lin89] Lin, Y. -B., Lazowska, E., Baer, J. -L., *Parallel Trace-Driven Simulation of Multiprocessor Cache Performance: Algorithms and Analysis*, University Of Washington, Department Of Computer Science. Technical Report 89-07-06, July 24, 1989

[Min89] Min, S. L., *Memory Hierarchy Management Schemes in Large Scale Shared-memory Multiprocessors*, PhD thesis, University of Washington, Department Of Computer Science. August, 1989

[Parn72] Parnas, D. L., On The Criteria To Be Used In Decomposing Systems Into Modules, *Communications Of The ACM. Vol 15, Number 12*, pages 1053-1058, December 1972

[Patt85] Patterson, D. A., Reduced Instruction Set Computer. *Communications Of The ACM. Vol 28, Number 1*, pages 8-21, January 1985

[Sche87] Scheurich, C., Dubois, M., Correct Memory Operation of Cache-Based Multiprocessors, *1987 ACM 0084-7495/87/0600-0234 00.75*, pages 234-243, 1987

[Seit85] Seitz, C. L., The Cosmic Cube, *Communications Of The ACM*, Vol 28, Number 1, pages 22-33, January 1985

[Seit89] Seitz, C. L., Seizovic, J. Su, W. -K., *The C Programmer's Guide to Multicomputer Programming*, Caltech Computer-Science Technical Report, Caltech-CS-TR-88-1, Revision 1, 17 April 1989

[Sene73] Seneta, E., *Non-Negative Matrices*, Wiley, New York, 1973.

[Simm90] Simmons, M. L., Wasserman, H. J., *Los Alamos Experiences with the IBM RISC SYSTEM/6000 Workstation*, Los Alamos National Laboratory, Los Alamos, New Mexico, LA-11831-MS, UC-905, March-1990

[Squi90] Squillante, M. S., Lazowska, E. D., *Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling*, University Of Washington, Department Of Computer Science. Technical Report 89-06-01, February 1990

[Swea85] Sweazey, P., CACHEPHOBIA: The Fear Of Cache Memories, *Computer Design*, 10 December, 1985

[Tane81] Tanenbaum, A.S., *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1981

[Varg62] Varga, R.S., *Matrix Iterative Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1962

[Wang88] Wang, W.-H., Baer, J.-L., Levy, H. M., *Organization and Performance of a Two-level Virtual-Real Cache Hierarchy*. University Of Washington, Department Of Computer Science., Technical Report 88-11-02, November 10, 1988

[Wan89a] Wang, W.-H., Baer, J.-L., *Efficient Trace-Driven Simulation Methods for Cache Performance Analysis*, University Of Washington, Department Of Computer Science., Technical Report 89-09-02, September 12, 1989

[Wan89b] Wang, W., *Multilevel Cache Hierarchies*, PhD thesis, University Of Washington, Department Of Computer Science., September, 1989

[Wils87] Wilson Jr., A. W., Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors, *Proc. 14th Symposium on Computer Architecture*, pages 244-252, 1987