AFIT/GCS/ENG/93D-1

VIRTUAL MEMORY MANAGEMENT AND VIRTUAL BUS OVERLOADING ON MULTIPLE CHANNEL ARCHITECTURES

THESIS

John N Armitstead Captain, USAF

AFIT/GCS/ENG/93D-1

Approved for public release; distribution unlimited

AFIT/GCS/ENG/93D-1

VIRTUAL MEMORY MANAGEMENT AND VIRTUAL BUS OVERLOADING ON MULTIPLE CHANNEL ARCHITECTURES

THESIS

Presented to the Faculty
of the Graduate School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Science

John N Armitstead
Captain, USAF

December, 1993

Approved for public release; distribution unlimited

Acknowledgements

My first thank you must go to Major Michael Gardner. He has the uncanny ability to make almost any situation bearable. Further, his selfless dedication to not only my success but that of all our classmates is an inspiration. My heartfelt thanks go also to his sweet wife, Pat, for her willingness to proofread this thesis.

I must also thank without reservation the combined efforts of my "team partner", Capt. John Reisner, and my advisor, Lt. Col. Tom Wailes. My success in this endeavor is a testimony of their devotion to academia and the individual. Their sometimes gentle and occasionally stern encouragements to me were the tail winds that kept me moving forward.

Finally, my love and eternal gratitude go to my wife and children. Kaylynn has worried and prayed over me more than even I know. Because of her I know that the hand of God has assisted me through this last season. My children, Joseph, James, Kristal, and Kathie always had a smile to help me remember why I began this undertaking.

Captain John N Armitstead

Table of Contents

List of Figures

List of Tables

Abstract

Today's computational environment requires the processing capabilities available only through parallel architectures. The bottleneck that limits the potential of parallel processing is communication between processors, memories, and other hardware devices. A proposed multiple channel architecture (MCA) utilizes tunable semiconductor lasers and fiber optic cables that serve as the communication medium between processor, memory, and I/O nodes. A memory management unit (MMU) was designed for use in the MCA. The design of the MMU was completely described and implemented in a multiprocessor simulator. A permutation-based interleaving (PBI) scheme was utilized to reduce the chance of memory access collisions. Virtual bus utilization, number of collisions, and message traffic patterns were studied under various amounts of overloading. Results show that it is possible to maintain processor efficiency while reducing demand for channel availability.

VIRTUAL MEMORY MANAGEMENT AND VIRTUAL BUS OVERLOADING ON MULTIPLE CHANNEL ARCHITECTURES

I Introduction

1.1 Background

Today's computing environment demands that results be produced at a rate faster than any single processor can achieve. Weather forecasting, chemical-reaction simulation, satellite imagery, and molecular interaction are only a few of the applications that require parallel computation. The answer to this problem is to create and improve multiple processor systems. One such system is the Multiple Channel Architecture (MCA) as proposed by Wailes [Wai92].

The MCA is unique in the multiprocessor arena due to the configuration of the switching network utilized between nodes (CPUs, Memories, I/O devices, etc). When compared to other possible switching techniques, the MCA has greater extensibility--the capability of extending a network architecture to accommodate an arbitrary number of nodes. For example, adding a node to an NxN cross-bar network quadratically increases the number of switches needed. Additionally, the number of nodes in a generalized-cube network must be a power of two. This means that any desired increase to an existing generalized-cube requires the network to double in size. Besides extensibility, other traits that exist in the ideal interconnection strategy include minimal diameter (longest path from node to node in terms of connections), minimal degree (number of I/O connections per node), and high simultaneity (the fraction of nodes that can transmit data simultaneously).

Partitionability, meaning that the network can be partitioned while guaranteeing that any partition of the network has all of the capabilities of the complete network, is also desired. The ideal network would also be non-blocking, so that it is possible to make a connection to any available resource. Finally such a network should maintain a uniform traffic distribution over the channels so that bottlenecks can be avoided [Wai92].

The MCA has high extensibility; it is able to easily add or remove nodes of any kind. This is possible because of the nature of the optical transmission medium and the use of a passive-star coupler. The passive-star coupler evenly distributes light energy entering via an input fiber into all of the connected output fibers. By connecting all laser transmitters to the input fibers and all receivers to the output fibers, it is possible to transmit from any source to any receiver. These transmissions have a maximum path length (diameter) of two: from transmitting node to passive-star coupler, and from passive-star coupler to receiving node.

Passive star couplers are available with up to 64 inputs and 64 outputs (64 x 64). It is possible to create couplers with any N x N configuration by connecting multiple smaller n x n couplers together. The laser power division over these composite couplers is almost ideal [Wai92]. Linke has shown that this type passive star coupler configuration can handle an almost unrestricted number of fiber optic cables when the laser sources have an output of more than 1 mW, and the sensitivity of the receivers exceeds 100 photons/bit [Lin88]. The laser devices described by Wailes for use in the MCA have multiple milliwatt power output, and the receivers he proposes have been shown to have sensitivity surpassing that required [Wai92].

The signal carried over the fiber optic cables will be generated by tunable lasers. At any given time the receivers in each node are tuned to certain assigned frequencies and signals intended for any particular node will be broadcast over that node's current frequency. Because of the nature of frequency division multiplexing used in the MCA

transmissions, it is possible to transmit messages on all possible frequencies at any specific time. This is analogous to the multiple signal environment used by television and radio. This broadcast capability gives the MCA high simultaneity and a non-blocking environment. Nodes within the MCA can have a degree as small as two, a transmission line to the passive-star coupler and a receipt line from the coupler is sufficient. Because the connection from any node to the passive-star coupler is independent from all other node to coupler connections, the MCA is arbitrarily partitionable without loss of capability in any of the partitions.

1.2 **Problem**

The architecture proposed by Wailes was described at a low level of detail, however, the specifics of individual node operations was not the focus of his work. One major component that needing detailed description is the memory management unit (MMU). The MMU is a part of the CPU node that translates the virtual address used within the processor to a physical address that exists somewhere within the memory system in the multiprocessor. Design of the MMU entails decisions on two major fronts: data block configuration within the memories, and interleaving methods used to distribute data among the various memory nodes in such a way as to achieve a more uniform distribution of traffic.

The baseline simulator created by Wailes during his study has the limited capability of running only a single process at any given time. The simulator was modified during this work to be able to simulate execution of multiple tasks simultaneously.

Another unique feature available on the MCA that was not explored in Wailes proposal is the ability to "overload" the communication channels. This is the capability to have traffic for more than one destination on a single bus. A "virtual bus" on the MCA

correlates to a laser frequency (channel), not to a physical communication line. Thus, changing the assignment of nodes to buses is accomplished by simply changing the receiver's tuner. In most multiprocessor architectures available today, transmission of data to a node is accomplished by setting the electrical switches in the network to eventually connect with the line that is physically attached to the node. Due to the nature of the transmission medium within the MCA, it is theoretically possible to have more than one node tuned to a particular frequency for receipt of data. This tuning configuration will then have traffic for all nodes with receivers tuned to a single frequency on a single bus. The modifications allowing multiple processes also made it possible in the simulation to overload buses with transmissions from more than a single process.

It was the intent of this work to show that this channel overloading can indeed be accomplished, increasing the utilization of the bus frequencies and reducing the demand on the finite (albeit large) number of frequencies available in the laser bandwidth. Further, it was the goal of this research to determine what level of overloading becomes detrimental to the operational capacity of the architecture.

1.3 **Scope**

It is possible to implement various MMU design schemes and compare the results to determine the optimal paradigm for this architecture. However, the focus of this work was to determine the design of the MMU and its component decisions, as previously described, utilizing knowledge of the nature of the MCA and the results of those researchers doing studies specific to the components or techniques in question. Further, the new version of the simulator used had a shared data cache that was not available in the previous version. This change made an "apples to apples" comparison of the proposed interleaving scheme in the new version with the standard interleaving in the previous version virtually impossible. This thesis does not, therefore, attempt to show any

quantitative performance changes from the previous version applicable to the design and implementation of the virtual MMU within the simulator. However, a comparison of these two interleaving schemes, both utilized in the new version of the simulator, was accomplished.

The scope of the possible test cases used to illustrate the feasibility of overloading buses on the MCA is very large. For example, consider the following: number of CPUs ranging from 1 to 1024 in powers of 2 (11 choices), numbers of local and shared memory nodes also ranging from 1 to 1024 (121 choices), similar ranges of numbers of buses to be allocated to CPUs and local and shared memories respectively (1331 choices), choices of overlaying of buses noting that offsets need not be a power of 2 (over 6 million permutations), software packages to execute on the simulator (unquantifiable), and the sizes of problems for the executed software to tackle (also unquantifiable). Taking the product of all of the previous values gives the number of choices that can be made for each individual process executed on the simulator. For this research, the simulator was allowed to simulate multiple tasks greatly expanding the possible configuration space.

Many of these choices are, in fact, nonsensical and given the obvious inability to try them all, only a few select cases were scripted and evaluated. Some of these cases will test the MCA's performance at increasing levels of channel overloading, while other cases will attempt to emulate possible scenarios for actual MCA operation.

Another area of focus that was not covered in this work was a protocol to ensure coherence of the translation lookaside buffer (TLB) for the shared memories. Discussion of the need for a TLB along with the presentation of a need for a coherence protocol is given in Chapter 6. Shared TLB coherence was not addressed for two primary reasons. First, the simulator used during this study does not have the capability to swap data blocks into and out of memory as would normally be the case with actual hardware. Second, such

coherency schemes tend to be complex and could reasonably be the primary focus of an independent research effort [Mil90].

1.4 Overview

This thesis reports the studies made of the MCA MMU (i.e. interleaving, data block configuration and general virtual to physical address translation). It also details the choices made for the circuit design of the MMU for the architecture. It further reports on the results of the hypothesis that multiple nodes will be able to receive data over the same communications channel without degrading the operation of the machine.

The following chapter is a review of the current knowledge of MMU operation, including address translation and interleaving methods. Chapter three details the design choices made for the MCA and gives justification for those choices. Chapter four describes the implementation of the MMU address translation and interleaving scheme within an already existing multiprocessor simulator used by Wailes in his work. It also describes in detail the test cases chosen to show bus overlay capability and comparison of interleaving methods. The fifth chapter focuses on the results of the various scenarios tested during this research. The final chapter reprises the conclusions reached during the research effort and closes with some recommendations for further studies.

II Literature Review

2.1 The Memory Management Unit

The need for memory management has emerged as multitasking and multiuser systems have been developed throughout the history of computing [Mil90]. Memory management is the mapping of virtual addresses to physical addresses. That is, mapping an address internal to a software process to one that describes the actual physical location of the data in memory. It is the job of the Memory Management Unit (MMU) to handle this virtual to physical address mapping.

The MMU is closely tied to the processor element in a multiprocessor system. Because the MMU performs the mapping or translation of all virtual addresses into physical addresses, design of the unit entails the design of a translation method. The design of this method is highly dependent upon the configuration of the data within the memory. This configuration is the result of a combination of block composition and subblock interleaving methods.

The ideal virtual to physical address transformation has several desirable characteristics. First, the translation must be made in one step. This one step generally includes concatenation of a number of low-order bits from the virtual address with a frame address retrieved from a look-up table using the high-order portion of the virtual address. The ultimate mapping scheme also will allow every memory request to be acknowledged by the memory units in minimal time. This means minimal waiting time until a previous request has finished.

2.2 Data Block Composition

Data is generally stored within memory systems using a blocked structure. The common management schemes that divide a memory address space into blocks are called segmentation and paging [Den70].

2.2.1 *Segmentation.* A segment is a contiguous variable-length linear array of addresses within the memory space. This variability has the advantage of being able to completely store any size data structure, up to the limits of the physical address space. A primary disadvantage of segmentation, however, is external fragmentation. This occurs when, because segments are not fixed in size, unused fragments or sections of memory tend to develop between segments after a series of allocations and deallocations. Figure 1 shows an example of external fragmentation. A minor disadvantage of segmentation is that the length of the segment must be recorded so that cross checks can be made during address translation to ensure the desired data item is within the confines of the segment.

2.2.2 *Paging.* A page is a fixed length contiguous linear array of addresses within the memory space. Since pages are all the same size, external fragmentation problems that occur with segmentation are no longer a concern. Furthermore, it is not necessary to maintain information regarding the length of the pages as they are all the same. Virtual address to physical address translation is also easier. This is explained in detail in Section 2.5.2. One disadvantage to paging, however, is that each process being executed may have one or more blocks of data that do not completely fill the allocated space. This is known as internal fragmentation. Figure 1 also shows how internal fragmentation could appear within memory.

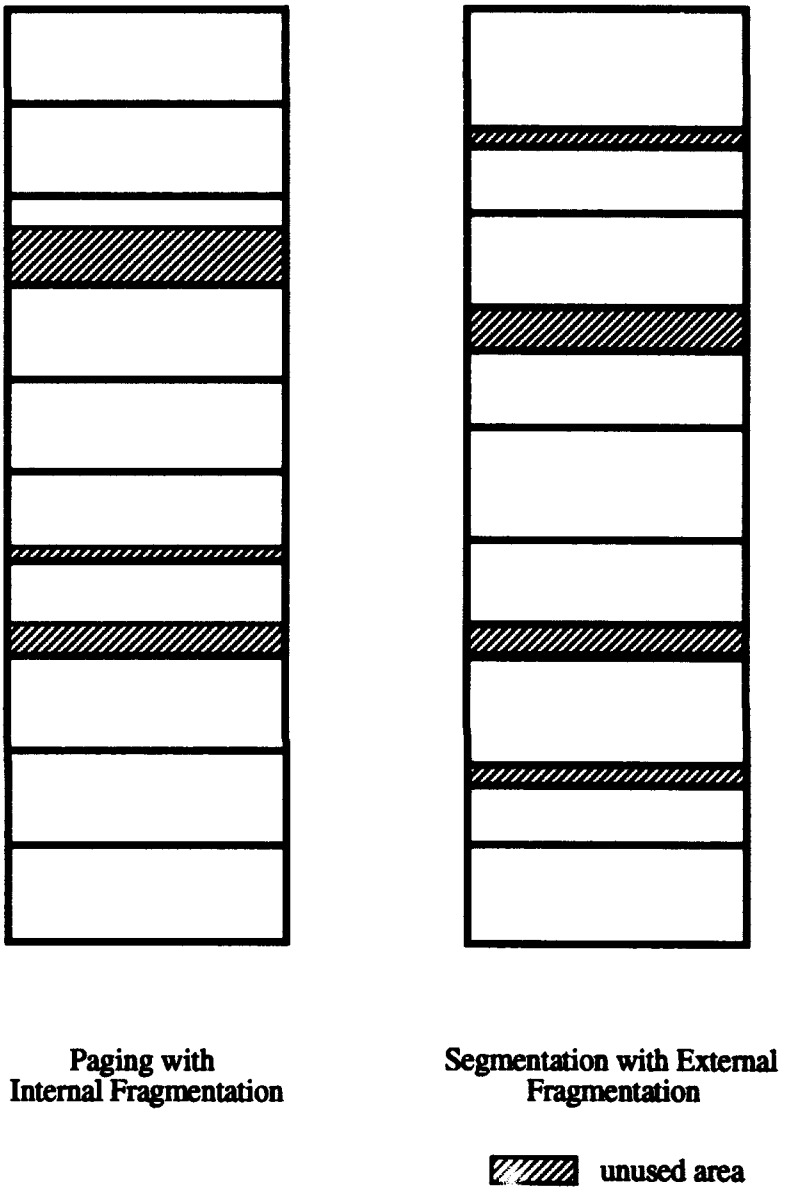


Figure 1 Fragmentation in Paging and Segmentation

2.2.3 *Segmented Pages/Paged Segments.* It is possible to combine segmentation and paging into one implementation, thereby accruing the advantages of both. This hybrid also tends to diminish the cumulative effects of the internal and external fragmentation of the blocks of data.

2.3 Interleaving of Memory Blocks

A technique that is commonly used in parallel machines with distributed memory is interleaving [War92][Bal86][Edl85][Pfi85]. A data block is partitioned into subblocks and these subblocks are assigned to successive memory units. Interleaving of memory blocks has two distinct advantages. One is that access to an entire block can be done in parallel over the units to which it is assigned when multiple buses are available for transmission of data. Another advantage is a reduction in the chance of collision when several processor units running in parallel are accessing different subblocks within the same block of data. Either of these advantages will increase the total throughput of a parallel system. [Kim91]

This method can also be used on mass storage devices (disks). A good description of disk interleaving is given by Kim [Kim91]. Further references to memory in this paper will not explicitly mention disk, but the concepts discussed are as valid for multi-disk storage as they are for multi-memory storage.

Within the realm of interleaving, there are two standard methods of assigning subblocks to memory units and two methodologies involved in assigning the subblocks to physical locations within the memory units.

2.3.1 *Synchronous vs Asynchronous Interleaving.* The first method of locating a subblock within a memory unit is termed synchronous interleaving [Kim91]. In this method each subblock is assigned the same predetermined location within its respective

unit. The most common formula for determining this location is taking the integer quotient of the physical address and the number of memory units. Thus, for a system with N memories, the first N subblocks will be in location zero of the respective memory units; the next N will be in location 1 and so on. Asynchronous interleaving, as its name implies, allows placement of subblocks in memory units without regard to where other subblocks have been placed. An example of synchronous interleaving is shown in Figure 2. Asynchronous interleaving is demonstrated in Figure 3.

Both synchronous and asynchronous interleaving deal with the "vertical" placement of data blocks, where vertical placement refers to the actual location within a memory module. Three other interleaving methodologies deal with the "horizontal" placement of data. This is the determination of which of the memory units associated with the particular data block will contain the desired subblock.

2.3.2 *Standard Interleaving.* The first and most basic of the three horizontal methods is a standard or 0-skew storage. For example, within n modules, module 0 will contain subblocks 0, n, 2n, 3n, ...; module 1 will contain subblocks 1, n+1, 2n+1, ...; module 2 will contain subblocks 2, n+2, 2n+2, 3n+2, ...; and so on with module n-1 containing subblocks n-1, 2n-1, 3n-1... The memory bank number is computed by taking the modulus of the data address and the number of memory modules. For example, subblock 43 would be found in bank 3 for standard interleaving over 8 memory banks. A diagram of sixty-four subblocks placed in eight memory banks using standard 0-skew interleaving is shown in Figure 2.

2.3.3 *N-Skew Interleaving.* Another scheme known as 1-skew interleaving is similar to the 0-skew method except that each successive row is shifted one module to the right (circularly) of the previous row. This scheme could be more generally viewed as n-skew interleaving where each row is shifted n modules to the right (again circularly) of the previous row. Computation of memory node for this interleaving is somewhat more

Data Block \ Memory Bank	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
2	16	17	18	19	20	21	22	23
3	24	25	26	27	28	29	30	31
4	32	33	34	35	36	37	38	39
5	40	41	42	43	44	45	46	47
6	48	49	50	51	52	53	54	55
7	56	57	58	59	60	61	62	63

Figure 2 Standard Synchronous Interleaving

Data Block \ Memory Bank	0	1	2	3	4	5	6	7
0	32	57	18	11	28	45	14	31
1	48	49	2	59	36	61	30	47
2	40	17	26	43	12	37	38	15
3	8	25	58	19	52	13	62	7
4	0	9	10	35	60	29	6	23
5	24	41	42	3	44	53	54	63
6	16	1	50	51	4	21	46	39
7	56	33	34	27	20	5	22	55

Figure 3 Asynchronous Interleaving

complex than for standard interleaving. First, vertical placement must be determined using integer quotient as given in the synchronous interleaving section. This number is multiplied by the degree of skewing with the result added to the virtual address. The memory node is determined by taking the modulus of this intermediate address solution and the number of memory banks. For example, if 3-skew interleaving were being done over 8 memory banks, address 43 would be found in memory bank 2. This is computed as follows: the integer quotient of 43 and 8 gives the vertical placement, 5; the product of the vertical placement and degree of skewing (3) is 15; this value added to the address (43) is 58; and 58 modulo 8 is 2. Diagrams showing this scheme for n=1 and n=3 are given in Figures 4 and 5 respectively.

2.3.4 *Permutation Based Interleaving*. A third scheme [Soh93] does not use skewing, but instead uses a pseudo-random permutation based interleaving (PBI) to mix up the blocks among the memory modules. This is described as a pseudo-random scheme because the permutation pattern repeats after M^2 subblocks, where M is the number of memory banks. This scheme utilizes an XORed Boolean product of a binary matrix with the virtual address to determine the memory unit or bank where the particular memory block is contained. A graphical example of PBI from [Soh93] is given in Figure 6.

These last two schemes, n-skew and PBI, are used to try to reduce the number of conflicting accesses. An access is in conflict if it is a request made to a memory unit that is currently busy handling another request. The conflicting request must then wait for any previous accesses to be completed. Two common causes of conflicting accesses are multiple requests to a shared data block by separate processors, and the stride of accesses not being relatively prime to the number of memory nodes. Stride is the incremental value between successive accesses to specific array elements within a processor. For example a stride of 4 would have accesses to array elements 0, 4, 8, 12, etc.

Data Block \ Memory Bank	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	15	8	9	10	11	12	13	14
2	22	23	16	17	18	19	20	21
3	29	30	31	24	25	26	27	28
4	36	37	38	39	32	33	34	35
5	43	44	45	46	47	40	41	42
6	50	51	52	53	54	55	48	49
7	57	58	59	60	61	62	63	56

Figure 4 1-Skew Interleaving

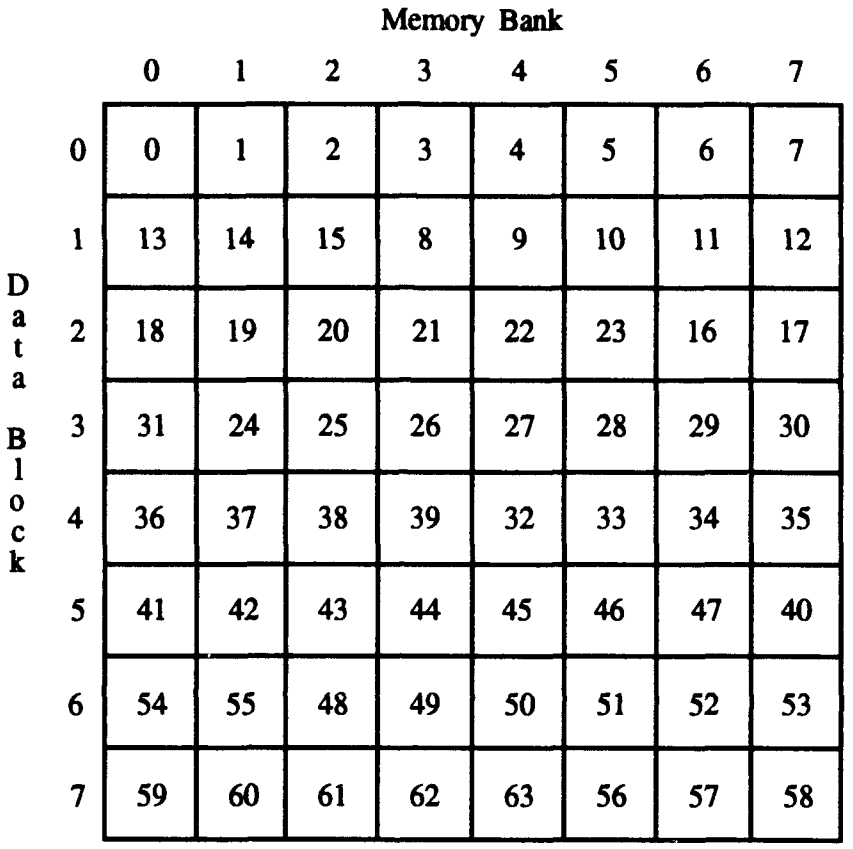


Figure 5 3-Skew Interleaving

Data Block \ Memory Bank	0	1	2	3	4	5	6	7
0	0	7	4	3	1	6	5	2
1	8	15	12	11	9	14	13	10
2	22	17	18	21	23	16	19	20
3	30	25	26	29	31	24	27	28
4	34	37	38	33	35	36	39	32
5	42	45	46	41	43	44	47	40
6	52	51	48	55	53	50	49	54
7	60	59	56	63	61	58	57	62

Figure 6 Permutation-Based Interleaving

2.4 Block Protection

Another job of the MMU is to ensure that attempted accesses are allowed. With any block that is available to a process, there are protections that have to be followed. Generally, these protections are enumerated as read only and read/write. In a distributed memory system with multiple processors and multiple tasks, the possibility of more than one processor accessing a single block of data also exists. This leads to the necessity of maintaining other protection information regarding the shared/private status of the block in question.

2.5 Conclusion

The design of the MMU entails decisions in two major areas: data block configuration and interleaving scheme. The two types of data blocks are pages and segments. The address translation steps used within the MMU are dependent upon the type of data block. There are five different types of memory interleaving: synchronous and asynchronous deal with the vertical (intra-memory bank) placement of data subblocks; while standard, N-Skew, and permutation based schemes deal with horizontal (inter-memory bank) placement.

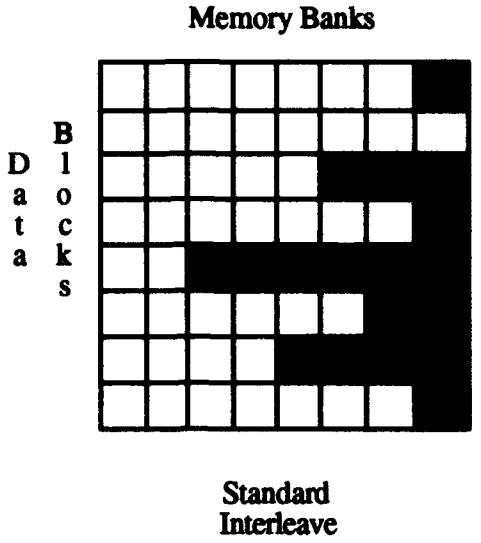
III Implementation

3.1 The MMU for the MCA

Design of the memory management unit (MMU) for use in the Multiple Channel Architecture (MCA) will be based on techniques described in Chapter 2. The data blocks are configured as pages, with the pages broken into subblocks for interleaving. The subblocks are distributed over the memories using a synchronized permutation-based interleaving (PBI) scheme. Block protection includes checks for read only, read/write, private, and shared designations.

3.2 Paging in the MCA

A good data block scheme for interleaved memory is paging. Using a paging scheme allows subblocks to be placed in all memories that are a part of the particular interleaving set. Some of the data subblocks may be empty (internal fragmentation), but this was found to be an insignificant problem within the MCA. If a standard interleaving scheme were to be used, the empty subblocks would all occur within the higher order memory banks, possibly causing disproportionate accesses between the higher number banks and the lower number banks. Because the lower numbered banks would have the majority of the data, they would be expected to receive a disproportionate number of the requests, increasing the chance of collision. With a PBI scheme, the empty subblocks are distributed more evenly over all of the banks so that there will be no increased possibility of collision within any given bank with respect to the location of the empty subblocks. Figure 7 shows an exaggerated case of internal fragmentation in two memory bank systems. The first shows how the empty subblocks are concentrated to the right when using standard interleaving. The second shows the same internal fragmentation as the first, but the empty



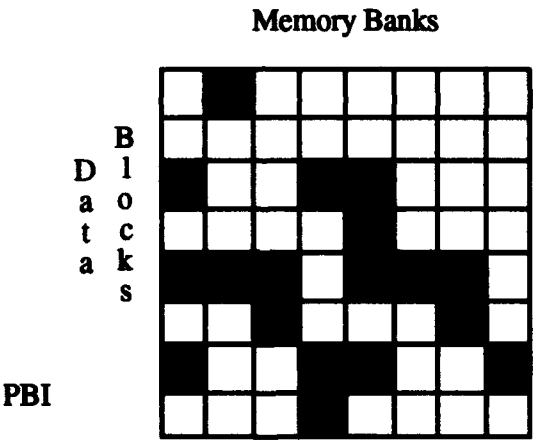


Figure 7 Internal Fragmentation and Interleaving

blocks are permuted among all of the memory banks. The permutation pattern used for this example was taken from the PBI example found in Figure 6 in the previous chapter.

Notice that in the standard example the first two memory banks do not have any empty subblocks and the next two memories have a single empty subblock. The 5th, 6th and 7th banks have two, three and four empty subblocks, respectively, with the 8th memory containing seven empty and only one used subblock. In the PBI example, the 6th and 8th banks have a single empty area, and there are two empty blocks in the 2nd, 3rd and 7th banks. The 1st and 4th banks show three empty spots, and the highest number of empty blocks in one memory, four, occurs in the 5th bank. This is still not the most ideal distribution possible, but it is an improvement over the distribution provided by standard interleaving.

3.2.1 *Subblock Size.* The size of the subblocks within the pages equal the larger of a private cache line or shared cache line. It is possible to have different line sizes for the private and shared caches. However, these line sizes are fixed [Rei93]. This independence of the private and shared line sizes does not cause a problem for the memories or interleaving scheme. The memory request packet constructed by the cache has the size and the beginning address of the needed line. This allows a memory to simply fill the request without requiring it to maintain knowledge of the cache line sizes.

Additionally, even though it is feasible a given memory unit will contain both private and shared data, the intermingling of the two possibly different sizes of subblocks will not cause external fragmentation as is possible with segmentation. All pages, by definition, will be the same size regardless of shared/private status. Each cache line size selected is constrained to be a power of two; even if they are different, one will always be a power of 2 multiple of the other. Thus, each subblock will contain either a single instance of the larger cache-line or a power of two instances of the smaller.

The difference in line sizes will not cause problems with the proposed interleaving scheme. The interleaving scheme used in the MCA, described in detail later, considers a subblock as a single entity of data. The address or identifier of a subblock is the address of the needed data item shifted right Log_2S bits, where S is the size of the larger cache line. This identifier will equal the tag used within the larger line size cache. The memory selection module simply uses this identifier as an input to the interleaving scheme to properly single out the memory bank containing the desired cache line.

3.3 MCA Interleaving

The MCA utilizes a synchronous, permutation based interleaving scheme. As briefly described in Chapter 2, a PBI scheme arranges the subblocks within a page in a pseudo-random order.

Synchronous interleaving is used in the MCA because the advantages of asynchronous interleaving apply solely to disk access. Kim describes how only seek and latency times can be affected by a choice between synchronous and asynchronous interleaving [Kim91].

Sohi gives an excellent review of the performance of his PBI compared and contrasted with the other more common interleaving methods, namely standard and n-skewed interleaving [Soh93]. As mentioned before, PBI is anticipated to distribute heavy access areas and light access areas more evenly over all of the memory banks. Further, the circuit implementing this scheme has only eight levels of gates to traverse (see section 3.3.3). This will make the PBI circuit faster than the circuit containing the adder necessary for an n-skew scheme. Therefore, his XOR-based PBI scheme was used within the MCA.

3.3.1 *How the MCA PBI works.* The subblock identifier that the MMU strips from the physical address is bit-wise ANDed to each row of a pre-computed binary matrix. The

content of this matrix is described later in more detail. Each of the resulting binary strings are internally XORed, giving a series of single-bit results. These bits are then concatenated, producing a binary representation of the number of the memory unit containing the desired cache line. This process is described by Sohi: (keep in mind that the MCA will access memory in cache line size chunks; Sohi is accessing individual words)

> If X is the N-bit address of the word and Y is the n-bit vector that represents the [memory] bank number [where 2^n=M memory banks], then Y is calculated as:
>
> $$Y = AX$$
>
> where A is an n x N matrix of 0's and 1's. The inner product is a logical inner product with the "multiplication (*)" being a logical AND operation and the "addition (+)" being a logical Exclusive-OR operation. Element Y_i of Y is, therefore
>
> $$Y_i = (A_{i,0}*X_0)+(A_{i,1}*X_1)+ ...[+(A_{i,j}*X_j)+...] +(A_{i,N-1}*X_{N-1})$$
>
> where X_j is the jth bit of the address [and $A_{i,j}$ is the jth bit of the ith row of the matrix] [Soh93].

The Exclusive-OR operation is essentially the same as a parity operation on the string; if the string has an odd number of 1's the result is 1, an even number of 1's give a result of 0. Sohi restricted his matrices to have exactly n rows, where n is the Log_2 of the number of memory banks, M, in the machine. Because the MCA is designed to have the capability of assigning varying numbers of memory banks to a process, the matrix A used must be capable of containing a variable number of rows. This turns out to be an easy requirement to handle; simply use the first n rows of the matrix. Of course the matrix should have sufficient rows to handle the largest possible value of n that could occur within a process being executed on the MCA.

3.3.2 *Composition of the Matrix.* The matrix described by Sohi is constructed of two submatrices AH and AL. AH is the high-order submatrix, consisting of the left most N-n bits of each row, where N is the width of the entire matrix. AL is the low-order

submatrix, made up of the right most n bits of each row [Soh93]. By using only the first n rows of the matrix, AL will always be a square submatrix regardless of the value of n. For example, if an architecture is using 32 bit addresses and a particular application is interleaving over 16 memory banks, the first 4 rows of the matrix would be utilized and AL would therefore consist of the rightmost 4x4 submatrix.

Sohi further states that the only constraints placed on the matrix to be used is that AL should be of full rank. With binary matrices, the determinant is taken with respect to the Boolean matrix multiplication operation. Sohi offers, and proves, a theorem justifying his claim that a full rank matrix in AL will provide a unique location for each element addressed in a PBI scheme. The reader is referred to his paper for elaboration of the proof. He also states that any pattern of binary digits within AH gives a valid permutation [Soh93]. The reader is also reminded that a matrix is considered to be of full rank if its determinant is non-zero.

As previously mentioned, the matrix used in the MCA must be flexible enough to handle multiple values of n. Therefore, the right most square submatrix of A must be of full rank regardless of the number of rows being utilized. One such matrix was discovered by starting with a full rank 1x1 AL, expanding it to a full rank 2x2, expanding that result to a 3x3, and so on. The MCA simulator used for this research has a maximum of 1024 (2^{10}) possible memories; thus, the matrix constructed has 10 rows. An inspection of each of the n-square sub-matrices (as n ranges from 1 to 10) in the upper right hand corner of the matrix given in Figure 8 shows that each one fits Sohi's requirements for AL. The MCA uses 48-bit addressing, so the matrices constructed for the MCA are 48x10 bits in size. As mentioned by Sohi, AH can contain any assortment of bits, and the one shown in Figure 8 was filled randomly. This matrix was not used for any of the testing for this work. Other matrices were derived and utilized in the test cases. These other matrices, and the

motivation for their use is given in Chapter four. If standard interleaving is desired the matrix presented in Figure 9 will give the proper permutation.

```
1010001100010111001011011100011110011101010101
0100001010001010011010101001001010100110101011
0101010011010101001010110101010100101101010111
0010111010011111010101010101010010101001010101111
0100110101001100001001000100010000001001011111
0100010110001111010110101010100100000110101111111
1001010101100101001001011010100100101001011111111
0101101010100101001010010010101000101110111111111
0110101011101111001101111100010100101001111111111
1001110010010011011010011100010000010011111111111
```

Figure 8 A General Transformation Matrix

```
000000000000000000000000000000000000000000000000001
000000000000000000000000000000000000000000000000010
000000000000000000000000000000000000000000000000100
000000000000000000000000000000000000000000000001000
000000000000000000000000000000000000000000000010000
000000000000000000000000000000000000000000000100000
000000000000000000000000000000000000000000001000000
000000000000000000000000000000000000000000010000000
000000000000000000000000000000000000000000100000000
000000000000000000000000000000000000000001000000000
```

Figure 9 A Transformation Matrix for Standard Interleaving

3.3.3 *Implementing this PBI in hardware*. The hardware required to implement this scheme is easy to construct. A circuit diagram is provided in Figure 10. A VLSI design software tool was used to estimate the area required for the latches (480 D flip flops), the 490 AND gates, and the 470 XOR gates required to build the circuit. These components, without routing, require less than 41.1 million square micro-meters. A standard chip has 72.68 million square micrometers available. This leaves over 43% of the area available for routing, pads, inputs, outputs, and the logic to load the registers at start up. With careful placement of the routing and components, it is feasible that this circuit can be implemented on a single standard chip.

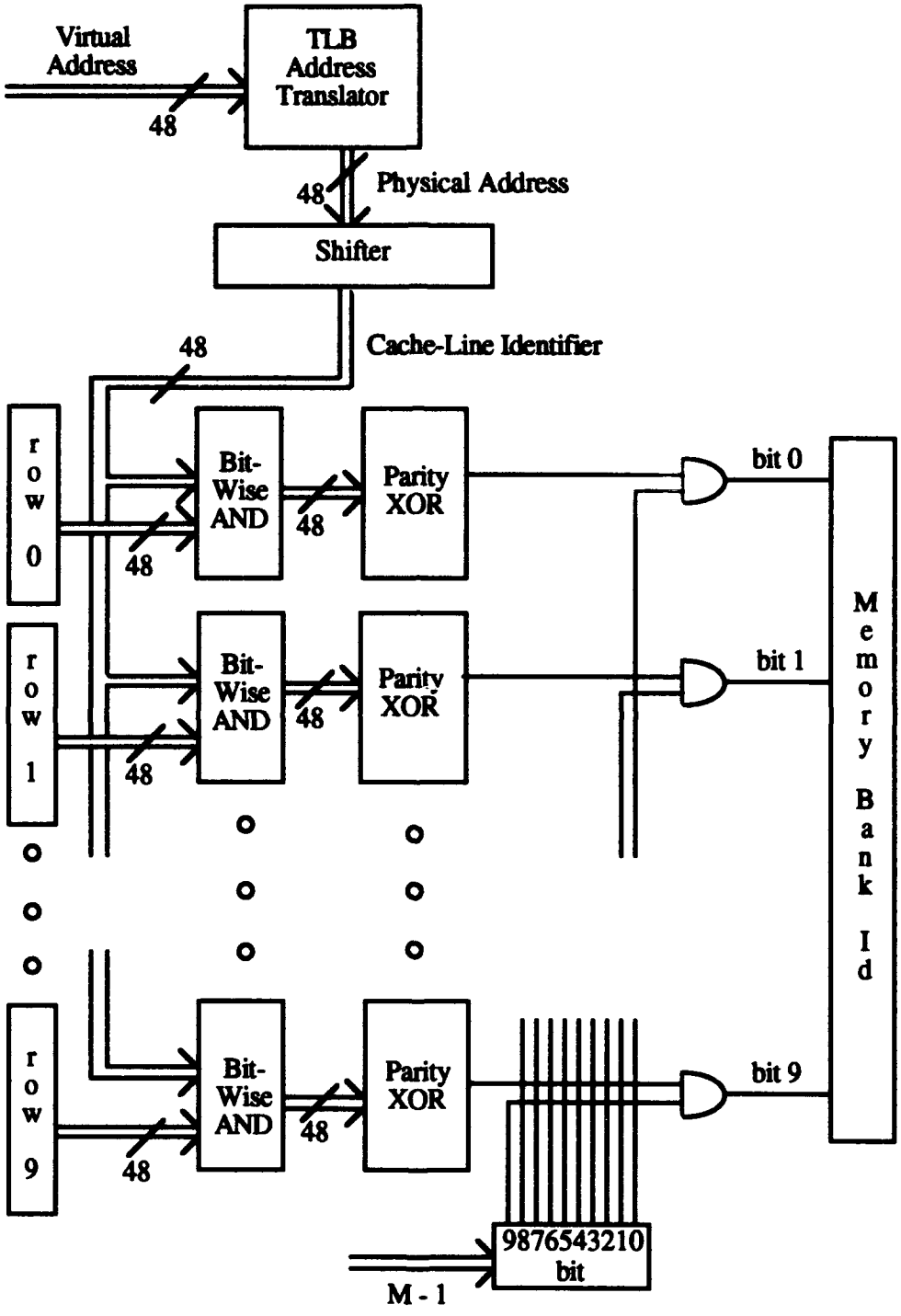


Figure 10 PBI Circuit

The address received from the virtual to physical translation function is shifted right an appropriate number of bits. The amount of shifting depends upon the size of the larger cache line (which is also the size of the data block). This determines the data block identification number. The shifted block identification number is passed simultaneously to ten sets of bit-wise AND gates, one set for each row of the matrix. The second input for each of the sets of AND gates is the respective row of the matrix. The matrix rows are kept in latches within the MMU. These latches are loaded at boot or process start-up time with a pre-computed matrix consistent with the desires of the user.

The output from each set of AND gates is passed through a set of XOR gates. Each set of XOR gates performs a parity function, giving the resulting bit to be used in the memory node identifier. As demonstrated by Sohi, the i^{th} row of the matrix is used to construct the i^{th} bit of the memory bank number. It is important to note that only the low-order Log_2M bits (where M is the number of memory banks) of the resulting number are used. This restriction is identical to only utilizing the first n rows of the matrix as described before, and allows us a range of resulting values from 0 to M-1. Selection of the proper bits to use is done easily; each bit of the XOR results is ANDed with the corresponding bit of the binary representation of the value M-1.

The result of these bit-wise operations is then used in a table look-up to determine the frequency of the channel for the selected memory. The frequency and bank number are passed back to the cache to be used to properly transmit a cache miss packet to memory if necessary. The frequency is used to tune the transmitter, and the bank number is included in the packet header so the various nodes receiving the packet (there may be more than one memory, and possibly even CPUs or I/O devices on the same channel) will be able to determine if they should accept or ignore the packet.

3.4 Multiple Nodes per Channel

A second focus of this research effort provides the ability to load channels with more than one node to determine a threshold loading of channels before performance degrades. Thus, it was necessary to rewrite the simulator to allow more than one process to run at any one time. In addition, overlapping the nodes from more than one process over the same channels was desired. The interleaving scheme would hopefully allow the accesses to be sufficiently random over the channels maximizing channel utilization while minimizing the degenerative effects of collisions on the channels. Descriptions of the changes to the code are given in Chapter four.

3.5 Conclusion

The MCA employs pages interleaved over the memory banks using a permutation-based interleaving scheme. The permutation of the data blocks is based upon a specialized binary matrix. The product of the data block address, or identifier, and the matrix with respect to the binary matrix multiplication operation gives the identifier of the memory bank containing the data.

IV Experiment Setup

4.1 MCA Simulator

The software package designed to emulate the multiple channel architecture (MCA) is called "smpcbs." Following good structured design practices, its main subroutines emulate the CPUs, memories, buses, and caches. The version of "smpcbs" developed prior to the beginning of this research effort (the "baseline" version) has the capability of emulating execution of a single process on the MCA using a variable number of CPUs, memories, and buses. Within the context of the configuration of the virtual machine, the caches are part of the processor elements or CPUs.

It is possible to overlap the buses in the baseline version; it is also possible to overlap the memories. In other words, the present version has the capability of assigning more than one CPU to a CPU bus, more than one local memory to a local memory bus, and more than one shared memory to each shared memory bus. It is further possible to assign buses to more than one kind of node. That is, a bus could carry traffic destined for CPUs along with traffic intended for local memories. A bus could carry traffic for local and shared memories, and could even be designated to carry traffic for CPUs along with both local and shared memories. With regards to memory overlapping, it is possible to assign more than one CPU per private data memory or more than one private data memory per CPU. All node assignments in the simulator must be powers of two. The simulator simply divides the larger group of nodes among the smaller group. It is further possible to designate memory banks as both private and shared using the offset designators in the start up parameter list.

The baseline version of "smpcbs" utilizes a standard interleaving scheme to determine memory banks for both local and shared memory accesses. Caching in the baseline version is restricted to local memory only. Shared memory data is not cached.

4.2 **Modifications to "smpcbs" - Interleaving**

Implementation of the XOR permutation-based interleaving (PBI) scheme explained in Chapter 3 was a relatively straightforward task. A routine, "get_memory_node", was created which accepts a physical address and returns a memory bank number. Different processes may have differing numbers of shared memories to interleave over, and may use different matrices to drive the XOR permutation. Because these parameters are not global in nature, they must also be passed to the XOR module. The algorithm used within "get_memory_node" follows the process explained in Chapter 3. First the subblock identifier (passed in as an integer) is converted into a binary string, an unnecessary operation within the physical implementation. The resulting address string is then bit-wise ANDed with each of the first $\log_2 M$ rows of the transformation matrix. During the bit-wise ANDing process, the variable intended to contain the parity result is toggled between 0 and 1 for each AND operation that results in a 1. This effectively produces the same result as an XOR parity operation on the resultant string. The resulting parity bits are concatenated to each other with the first result becoming the least significant bit and each succeeding bit being concatenated to the left (i.e., each new result bit becomes the most significant bit). Thus, there is one parity result bit concatenated for each bit-wise ANDing between a row of the matrix and the address string. When the ANDing, parity computation, and concatenation is complete for all needed rows, the integer constructed by the concatenation of the parity result bits is the memory number and is passed back to the calling module. A copy of the source code is found in Appendix A.

As mentioned in Chapter 3 the actual hardware implementation keeps the rows of the transformation matrix in latches or registers within the PBI circuit; these are loaded at boot-up or during initialization of the configuration. Because it is very likely that the shared data will be interleaved over a different number of memory banks than the private memory, it becomes necessary to have the respective caches latch the appropriate value (number of memories - 1) to the MMU so that the proper number of lines of the matrix are utilized.

4.3 Modifications to "smpcbs" - Multiple Processes

The current state of the MCA is simulated, both in the baseline and in the version created for this research, using large data structures. There is a structure for the status of each CPU containing, among other things, its computation state (STALLED, RESTARTing, EXECUTing, etc), the next instruction to be handled, the registers, and so on. The memories, caches, and buses also have structures to maintain their state. The baseline "smpcbs" does not have a structure to maintain the state of the process. It simply uses a group of variables not formally contained within a structure to maintain information regarding numbers of nodes and buses, overlapping (as described previously), and copies of the object code and data. This was allowable because information was maintained only for a single process. With the need to run multiple processes on the MCA came the requirement to encapsulate this information within a structure. The ability to have a variable number of these structures maintaining information for all of the processes to be simulated at one time is also a significant requirement. This modification entailed placing all of the appropriate variables within a new structure that was appropriately called "process." The text editor's find-and-replace function located and modified all occurrences of the necessary variables enabling them to correctly identify themselves as a part of the new structure.

It was also necessary to identify any given node or bus with respect to its placement within the entire system and its associated process(es). The former is easy to do. The CPU, memory and bus structures are configured as arrays of structures. Thus, their placement within the entire system is simply their array identifier. Maintaining a node's placement with respect to the process it is executing necessitated the addition of a variable to the structure which supplied the number of the first of the group of respective nodes within the process. The relative number of a node is then obtained by subtracting the number of the first (or 0th) node from the system number of the node in question.

Another status structure that must be unique to each process is the BARRIER structure. In the baseline version, a single BARRIER was able to control the progression of all of the CPUs within the system. With the advent of additional processes, it quickly became evident that the single BARRIER was confusing as the two processes are independent. Therefore, the BARRIER structure is also included as a sub-structure within the process superstructure.

A further change that had to be made to the previous version was the loading of the process configuration parameters. In the baseline version, these parameters were supplied by the user on the simulator host machine's command line. With the need to be able to have multiple processes, it became necessary for parameters be read from an input file with one line of parameters for each process desired. The name of this file was supplied to the simulator via the command line. The simulator initialization module was modified to accept the configuration parameters from a disk file.

4.4 Test Cases

As mentioned within the scope section of Chapter 1, the testing on "smpcbs" focused on determining the validity of the hypothesis that it is possible to overlap bus assignments to nodes without degrading CPU efficiency. Two main sets of experiments

were eventually developed to show: 1) the relative regression of CPU utilization as the bus loading/overloading increased; and 2) that a careful multiple assignment of buses can be made without a significant effect on efficiency.

4.4.1 *Background.* Within each process on the simulator there are three contiguous bus "spaces." The spaces are assigned to the CPUs, local memories, and shared memories. This is done by assigning certain numbers of contiguous buses to each space and then specifying the numbers of the buses that will be the logical 0th bus for each bus space. This designation is made for each of the spaces in each of the processes to be run on the simulator.

An example may make this a little clearer. Let process 0 on the simulator be assigned four each of CPUs, local memories, and shared memories. The CPU bus space is given 4 buses beginning at bus 0; the local memories are given 4 buses beginning at bus 4; and the shared memories are given 4 buses beginning at bus 8. The bus assignments are portrayed in Table 1.

Table 1 A Single Process Bus Assignment Example

Bus	Assignment	Bus	Assignment
0	CPU 0	6	Lcl Mem 2
1	CPU 1	7	Lcl Mem 3
2	CPU 2	8	Sh Mem 0
3	CPU 3	9	Sh Mem 1
4	Lcl Mem 0	10	Sh Mem 2
5	Lcl Mem 1	11	Sh Mem 3

A second process is then initiated with 4 each of CPUs, local memories, and shared memories. Each of the spaces in this process (designated #1) is assigned 4 buses just as in process 0. However, the beginning bus number for each of the spaces is assigned as 2, 6 and 10 respectively. The bus assignments for this change are shown in Table 2. Even

though this was not one of the configurations used for testing, it shows how the buses could be assigned to single or multiple nodes.

Table 2 A Dual Process Bus Assignment Example

Bus	Assignment	Bus	Assignment
0	CPU 0 (Process 0)	7	Lcl Mem 3 (Process 0)
			Lcl Mem 1 (Process 1)
1	CPU 1 (Process 0)	8	Sh Mem 0 (Process 0)
			Lcl Mem 2 (Process 1)
2	CPU 2 (Process 0)	9	Sh Mem 1 (Process 0)
	CPU 0 (Process 1)		Lcl Mem 3 (Process 1)
3	CPU 3 (Process 0)	10	Sh Mem 2 (Process 0)
	CPU 1 (Process 1)		Sh Mem 0 (Process 1)
4	Lcl Mem 0 (Process 0)	11	Sh Mem 3 (Process 0)
	CPU 2 (Process 1)		Sh Mem 1 (Process 1)
5	Lcl Mem 1 (Process 0)	12	
	CPU 3 (Process 1)		Sh Mem 2 (Process 1)
6	Lcl Mem 2 (Process 0)	13	
	Lcl Mem 0 (Process 1)		Sh Mem 3 (Process 1)

Matrices following the design constraints given in Chapter 3 were constructed to test the simulator's ability to use different matrices for each process. Some of these are shown in Figure 11. Notice the 'D' and the integers constructed of the 1's in the matrix. This was done to help identify the matrices used in each simulation run. 'D' is for the default matrix and the numbers correspond to the numbered optional matrices available to simulation testers. Constructing these characters within the matrices did not degrade their validity; recall that any combination of 1's and 0's in AH will suffice [Soh93].

4.4.2 *Overload Stress Breakpoint Test.* The first of the two main test suites was designed to increase the loading on the buses and watch for significant degradation of CPU utilization. The test suite consisted of nine configurations of bus assignments beginning with one node per bus (a crossbar-like configuration) and ending with a single bus handling all traffic within the simulator (a global bus like configuration). In each of the nine executions, there were two processes being run, a 64x64 median filtering problem and a 128x128 matrix multiplication problem. Each process was assigned 64 CPUs, 64 local

```
111110100100111100111111111111100001110101010101
001111100110100100111111000011110000101010101011
010110101011110100000111000000111000010101010111
010011100100111100000111000000011100001010101111
111010010111001010001110000000011100101010111111
101011101101100101100111000000001110001010111111
010101010001001111000111000000011100010101111111
001011111011101000000111000000111000001011111111
010100010001000000111111000011110000110111111111
100100010000001000111111111111100001001111111111

101001111001011100101101110001111001110101010101
010011111000101001101010100100101010011010101011
100111111001010100101011010101010010110101010111
001111111001111010101010101010010101001010101111
100001111000000000100100010001000000100101011111
010001111000111101011010101010010000011010111111
100001111001000100100101101010010010100101111111
000001111000001010111001001010100010111011111111
001111111111001001001011110001010010100111111111
001111111111001000001001110001000001001111111111

010000111111000010010110111001111001110101010101
100001111111100000110101010100101010011010101011
010011100001110010010101010101010010110101010111
100000000111100010101001101010010101001010101111
000000000111100000010100010001000000100101011111
100000001111000010011010101010010000011010111111
010000011110000010010010110010010010100101111111
100001111000000001011100101010100010111011111111
100011111111111000010010111000101001010011111111
110011111111111000000000011100010000010011111111
```

Figure 11 More Examples of Transformation Matrices

memories, and 64 shared memories. Each run was given designations of the letters 'c' through 'k' respectively. In run 'c', each bus was assigned a single CPU or memory node. In run 'd', each bus was assigned a pair of like nodes, one from process 0 and one from process 1. In runs 'e' through 'j', the number of buses was cut in half from the previous run, doubling the number of assigned nodes per bus. In run 'j', a single bus handled all traffic destined to any CPU in both processes; a single bus also handled all traffic for local memories. A third solitary bus handled all traffic for the shared memories. In run 'k', these three buses were consolidated into one, handling all traffic for all 384

nodes in the system. Figures 12 through 14 graphically illustrate the first three of the nine configurations .

4.4.3 *Realistic Overload Test.* The second test suite also used the filtering and matrix multiplication programs. The numbers of nodes assigned were identical to the first suite. However, the point of this suite was to try to show a practical application of the use of bus overloading. Therefore, the overlapping of nodes to buses was done in a more selective and less forceful way. Each process was given 32 buses for the CPUs, 32 buses for the shared memories and eight buses for the local memories. It was anticipated that traffic to the private data (local memories) would be lighter than for the CPUs and shared memories. Therefore the local memory buses were given a higher load. A control run (designated 'x') was executed with each of these spaces independent of one another.

Two other runs were then performed. Because of the nature of parallel algorithms, it is often the case that the 0th processor must perform some serial task(s) while the other processors wait. It was felt that there would be negligible effect on total execution time of the algorithm when overlaying the local memory buses with the CPU buses for those CPUs that are waiting on the serial portion of the algorithm to complete. Therefore run 'y' had both sets of local buses combined with the higher number processor buses in the matrix multiplication process, and run 'z' had both sets of local buses combined in a similar manner with the median filtering CPU buses. In both cases the local memory bus sets did not overlap each other. Figures 15, 16 and 17 portray these configurations.

4.5 Test Case - Interleaving Methods

A comparison of standard interleaving memory access patterns and PBI memory access patterns was conducted by running two test executions, each with exactly the same configuration except for the binary matrix input to the PBI translator. These two cases were very similar to the baseline for the overloading tests. Each process contained 64 each

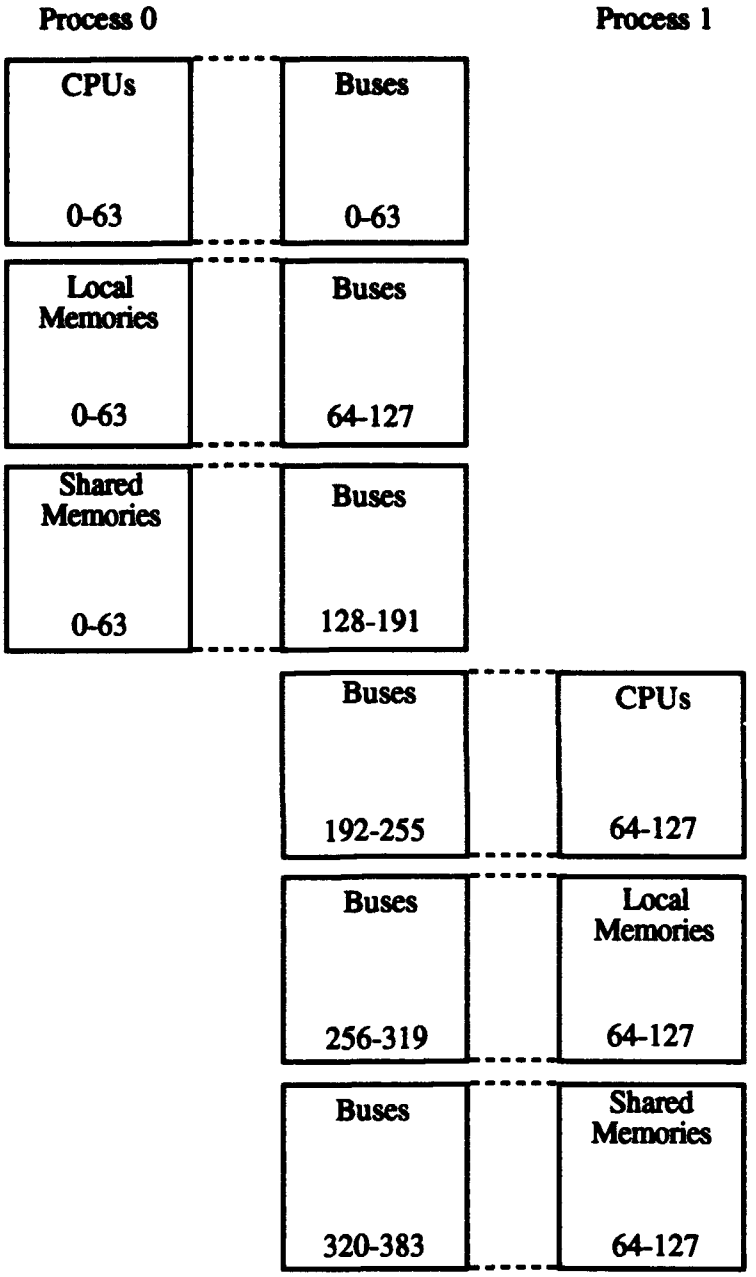


Figure 12 Test Case 'c'

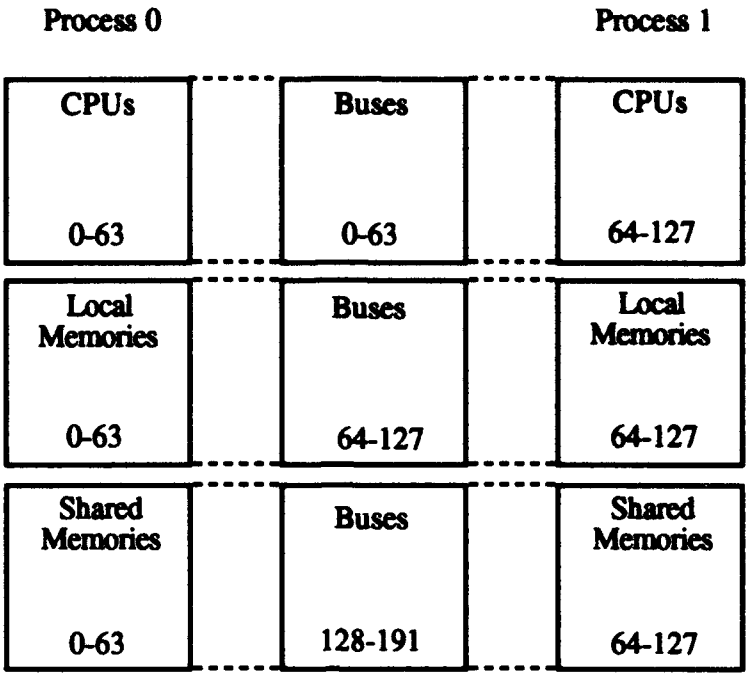


Figure 13 Test Case 'd'

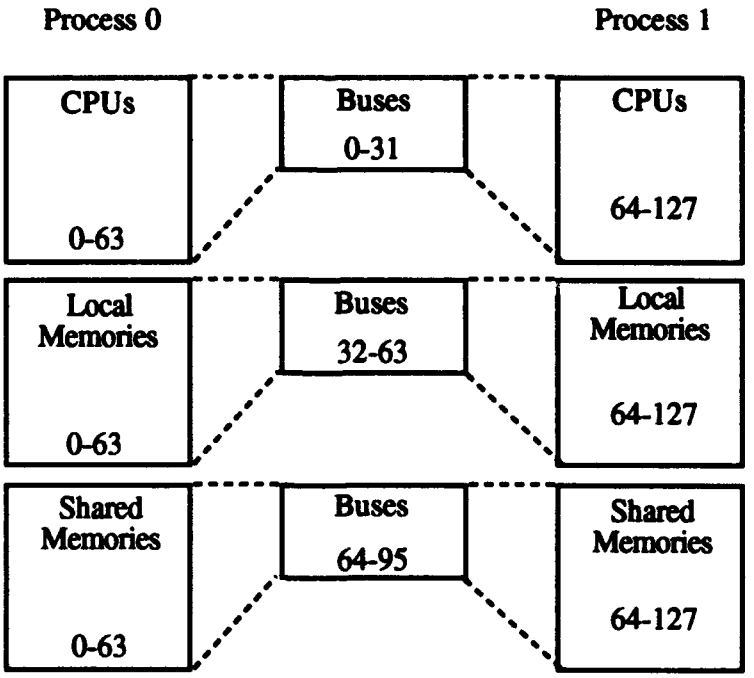


Figure 14 Test Case 'e'

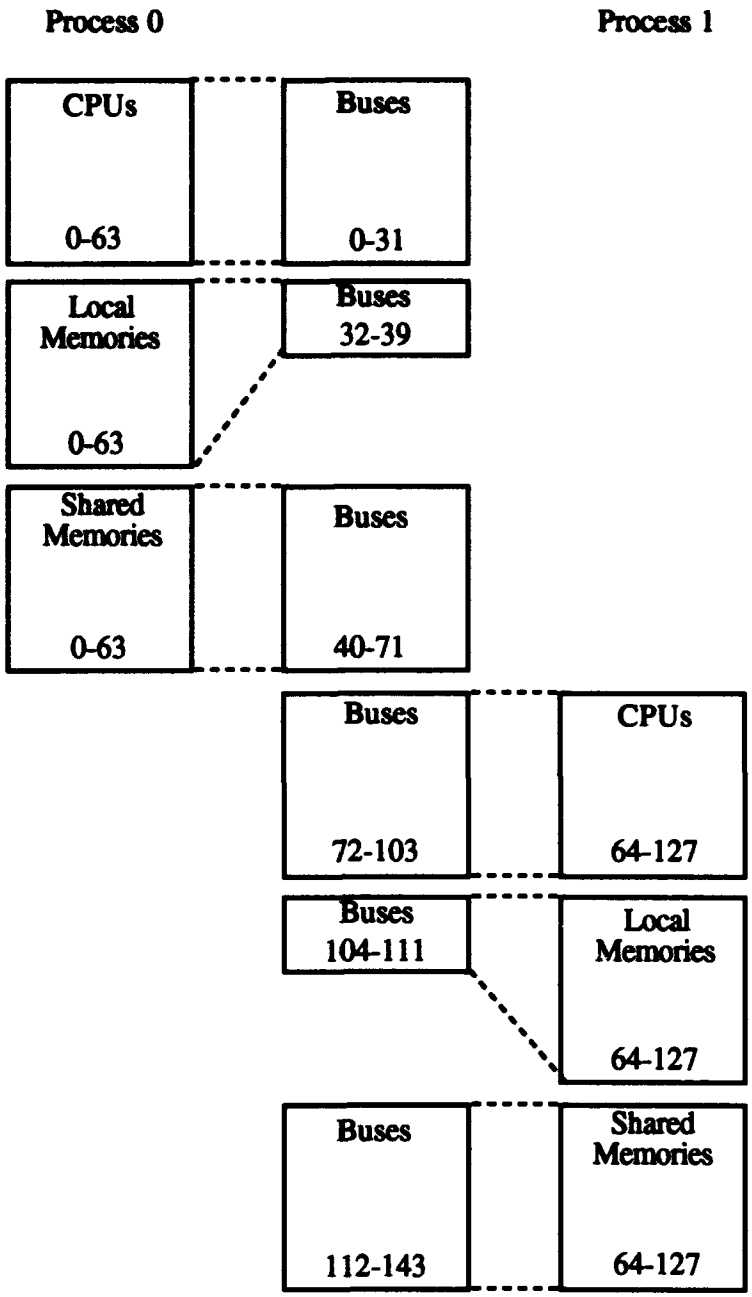


Figure 15 Test Case 'x'

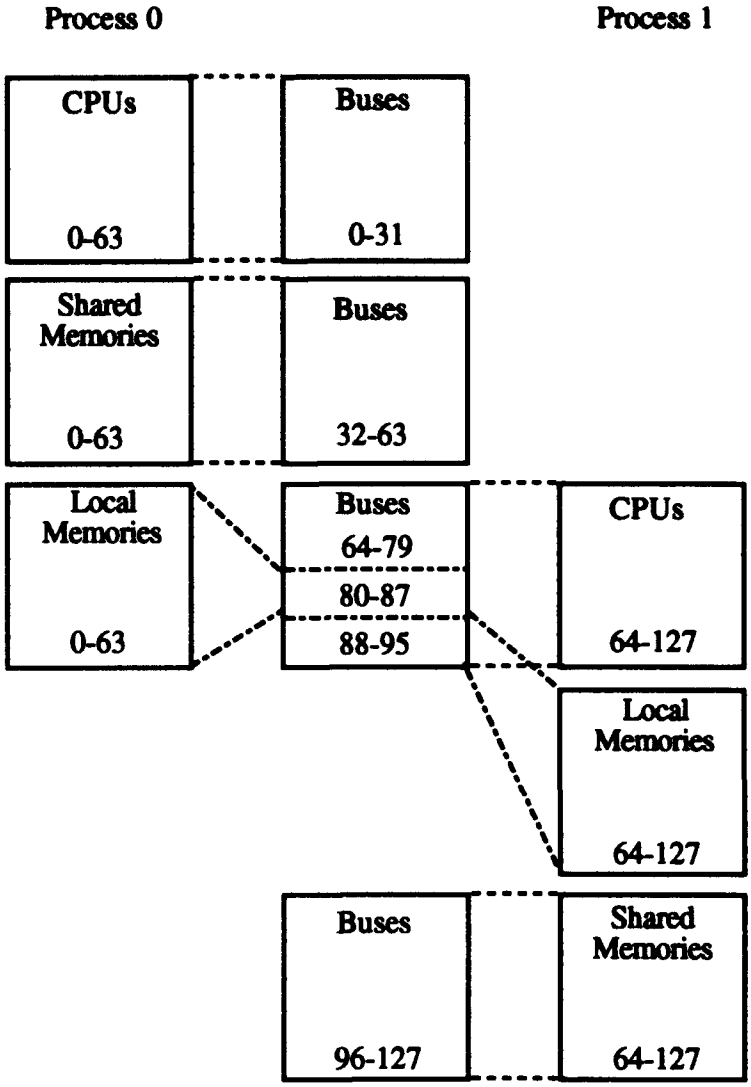


Figure 16 Test Case 'y'

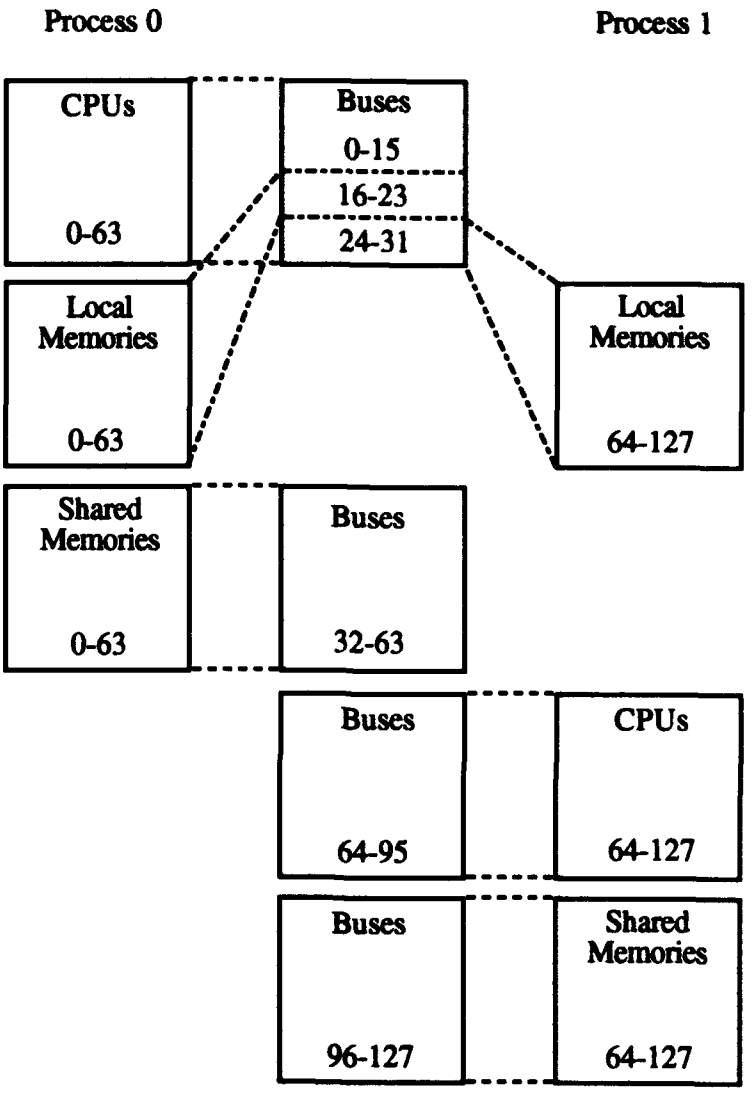


Figure 17 Test Case 'z'

of processors, local memories, and shared memories. As mentioned in Chapter 1, uniformity of bus traffic is an attribute of an ideal multiprocessor. To make an evaluation of this criterion easier, each of these 192 nodes had its own channel. The first run, designated 'a', utilized the matrix shown in Figure 9 in Chapter 3. This was the standard interleaving run. The second run, designated 'b', employed matrix number seven created for this research effort. This matrix is shown in Figure 18.

```
110010101010000011111111110001001001000101010101
110101011010110011111111110001010101111010101011
010010101101010000000011100101010010110101010111
111100110101110000000111001101100101111010101111
000010010001000000000111000110001000010010101111
110101101010100000011100101110101000011010111111
111101010111010000111000111010100101110101111111
101011100100000000111001101101100010111011111111
101110101101010000111000101011110010110111111111
100001001001110000111000101100001011111111111111
```

Figure 18 Transformation Matrix 7

4.6 Conclusion

The existing parallel simulator "smpcbs" was modified to utilize the PBI scheme detailed in Chapter 3. It was also modified to be able to run multiple processes with the ability to assign multiple processor nodes to singular bus channels. Several test cases were developed to determine the effect on CPU utilization of different configurations of bus assignments intended to progressively increase the bus loading. Other test cases were developed to evaluate CPU performance with a realistic bus loading scenario. Finally, a test case was developed to compare overall memory bank accesses between standard interleaving and permutation-based interleaving.

V Simulation Test Results

This chapter reviews and evaluates the results of the changes made to the simulator software. It presents verification of changes made as valid and accurate. Results from the test cases described in the previous chapter are also evaluated.

5.1 Design of the MMU

The design of the memory management unit (MMU) for the multiple channel architecture (MCA) presented in Chapter 3 shows that the hardware for the permutation-based interleaving (PBI) scheme can be implemented on a single chip. With only eight levels of gates to traverse, it will be faster than the adder required for N-skew interleaving. The implementation of this scheme in the simulator as described in Chapter 4 shows that the algorithm to translate unique addresses to unique memory bank numbers does indeed work properly. Snapshots of communications packets were examined to ensure that the memory banks identified during execution were indeed those that would be associated with the cache line containing the specific address. One of these snapshots is given in Figure 19. The destination address (labeled destin.addr), as shown in the example, was used as input to an independent XOR transformation program. This independent program had been

```
CPU[0] sending a packet (cache miss). Msg22. Contents follow:
Packet address is 4993536
sender is CPU, bus 0        source. addr = 0          source. node = 0
destin. is MEM, bus 4       destin. addr = 4164640  destin. node = 4
op is LOAD_LINE  ;ack is NONE
b'day = 2;  xmit_time = 2;  length = 32
cval = ival = 4993632  dvall[0] = 0.000000
sval = 76      ival = 4993632   dvall[1] = 0.000000  value.dval = 0.000000
backoff = 0;  active_index = 0;  next = 0;  sh = 0;
P_stat = ASSMBLD
```

Figure 19 Transmission Packet Snapshot

previously verified as correctly providing a valid permutation of data subblocks according to the transformation matrix utilized. The output from this program was checked against the memory node identified in the snapshot (labeled destin. node). In every case, the two results matched. Thus, it was determined that the PBI scheme algorithm implemented within "smpcbs" was providing correct results. This test did not provide analysis of the effectiveness of the permutation patterns. Such analysis is presented in Section 5.3

Several of the numbered transformation matrices introduced in Chapter 4 were also loaded for various processes. The capability to use different matrices for different processes was verified through simulator output. An example of the simulator report giving configuration data is provided in Figure 20. Visual inspection of the output confirmed that the desired matrices were associated with the proper processes. The output shown was also extremely useful for verification of the configuration of the individual processes. These outputs were routinely checked to ensure that the desired configurations were indeed utilized for the test cases.

5.2 **Results of Bus Overloading Tests**

The results derived from the overloading tests as described in Chapter 4 demonstrate that it is possible to assign multiple nodes to a single bus without seriously diminishing processor performance.

5.2.1 *Overload Stress Breakpoint Test.* The first suite of runs performed what could be termed a stress tolerance test. The baseline run, used to identify top possible performance, had a single node to single bus assignment plan. Further runs doubled the number of nodes assigned to each bus from the previous run. Refer to Chapter 4 for a detailed description of the configuration of the simulator for each test run. Snapshots, or histograms, of the system's performance were taken every 250,000 simulator clock cycles. The CPU utilization percentage for each processor during each histogram period was

```
mpcbs input for process number 0
        buflen=16               backoff_limit=10
        Ncpus=64                Cpu_offset=0
        nbuses_cpu=64                   Cpu_bus_offset=0
        nbuses_local=64                 Local_bus_offset=64
        nbuses_shared=64        Shared_bus_offset=128
        nmems_local=64,                 Local_mem_offset=64
        nmems_shared=64,        Shared_mem_offset=0
        cache_line_size=32,     sh_cache_line_size=32
        num_cache_lines=512,            sh_num_cache_lines=256
        cache_depth=2                   sh_cache_depth=4
        cache directory_size=0,         cache_flags=8244

        XOR Transformation Matrix:
                111110100100111100111111111111100001110101010101
                100111110010100100111111000011110000101010101011
                110100111001110100000111000000111000010101010111
                010011100101111100000111000000011100001010101111
                111010010110100101000111000000001110010101011111
                101011101010100101100111000000001110001010111111
                010101010010000111000111000000001110001010111111
                001011111010101000000111000000111000001011111111
                010100010000100000111111000011110000110111111111
                100100010001001000111111111111100001001111111111

mpcbs input for process number 1
        buflen=16               backoff_limit=10
        Ncpus=64                Cpu_offset=64
        nbuses_cpu=64                   Cpu_bus_offset=192
        nbuses_local=64                 Local_bus_offset=256
        nbuses_shared=64        Shared_bus_offset=320
        nmems_local=64,                 Local_mem_offset=192
        nmems_shared=64,        Shared_mem_offset=128
        cache_line_size=32,     sh_cache_line_size=32
        num_cache_lines=512,            sh_num_cache_lines=256
        cache_depth=2                   sh_cache_depth=4
        cache directory_size=0,         cache_flags=8244

        XOR Transformation Matrix:
                010000111111000010010110111001111001110101010101
                100001111111100000110101010100101010011010101011
                010011100001110010010101010101010010110101010111
                100000000111100010101001101010010101001010101111
                000000000111100000101000100010000001001010111111
                100000001111000010011010101010010000011010111111
                010000011110000010010010110010010010100101111111
                100001111000000001011100101010100010111011111111
                100011111111111000010010111000101001010011111111
                110011111111110000000001110001000001001111111111
```

Figure 20 Smpcbs Configuration Output

measured. As bus traffic increases, the time that a CPU will wait for a memory request packet to return is expected to increase, causing a decrease in CPU utilization. The graph in Figure 21 shows the decrease of CPU utilization for the main (0th) processor running the 128x128 matrix multiplication program. This was processor 64 in the simulation. The x-axis of the graph denotes the individual histogram periods. The y-axis denotes the different test runs. Run 'c' (minimum bus loading) is in the background with run 'k' (all traffic on a single bus) in the front. The z-axis shows the utilization of the CPU. Note that the graph slopes more steeply to the left (early histograms) than to the right. This is mostly due to the fact that the other process being simulated, a 64x64 median filtering problem, would terminate sometime between the beginning of the fourth period and the end of the sixth period. Evaluation of the slope for the first five runs (c - g) shows that CPU performance degraded quickly with each increase of bus traffic. This suggests that any arbitrary overloading of buses could seriously impact processor performance.

The graph in Figure 22 gives a percentage of collisions to transfers for the buses that could affect the performance of CPU 64 (relative CPU 0 in process 1). These were 1) the bus assigned to the CPU itself, 2) the bus assigned to the memory bank containing the CPU's private data, and 3)all of the buses assigned to memory banks containing shared data for process 1. The x-axis denotes these three bus sets; the y-axis is again the test runs; the z-axis gives the collision percentage. It is evident that the increase in the collisions on the shared memory and CPU buses is primarily responsible for the dramatic decrease in CPU utilization that occurred. The increase in local memory bus collisions is slight in the first few test cases suggesting a less significant contribution to the early degradation of CPU performance.

5.2.2 *Realistic Overload Test.* The second suite of tests were designed to show that a carefully planned overloading of buses could decrease the demand on the number of frequencies (or channels) needed without seriously downgrading the

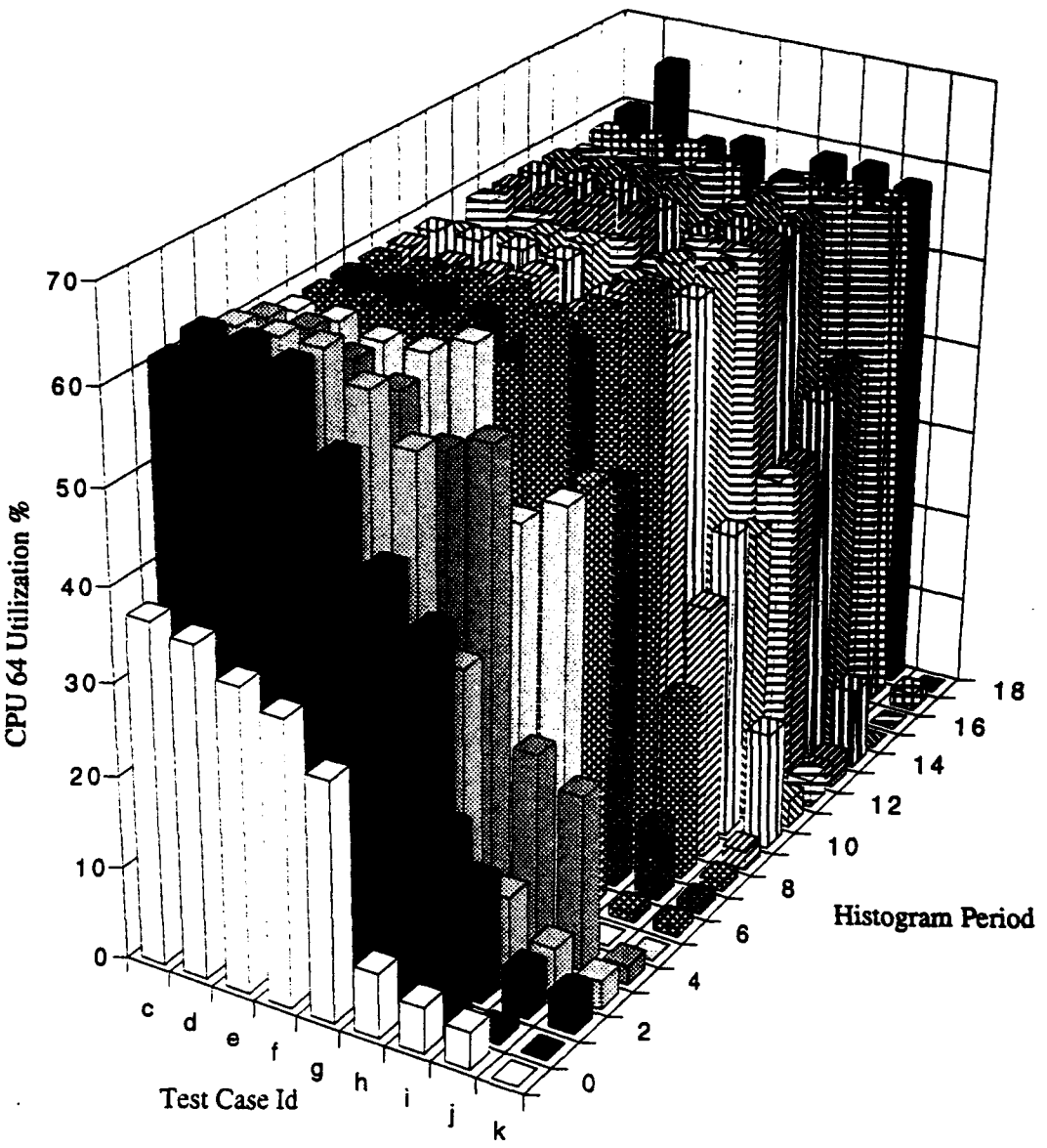


Figure 21 Stress Breakpoint Test--CPU 64 Utilization

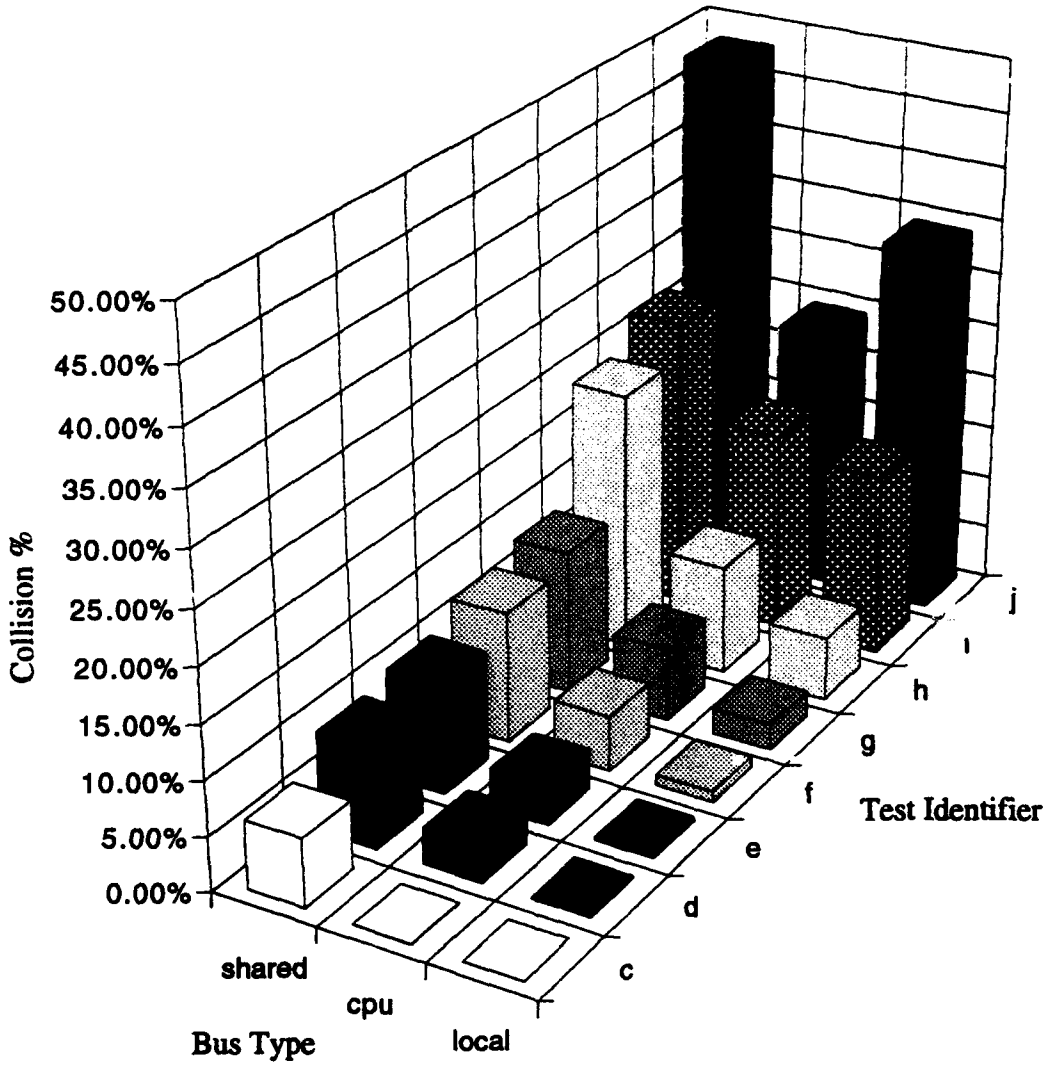
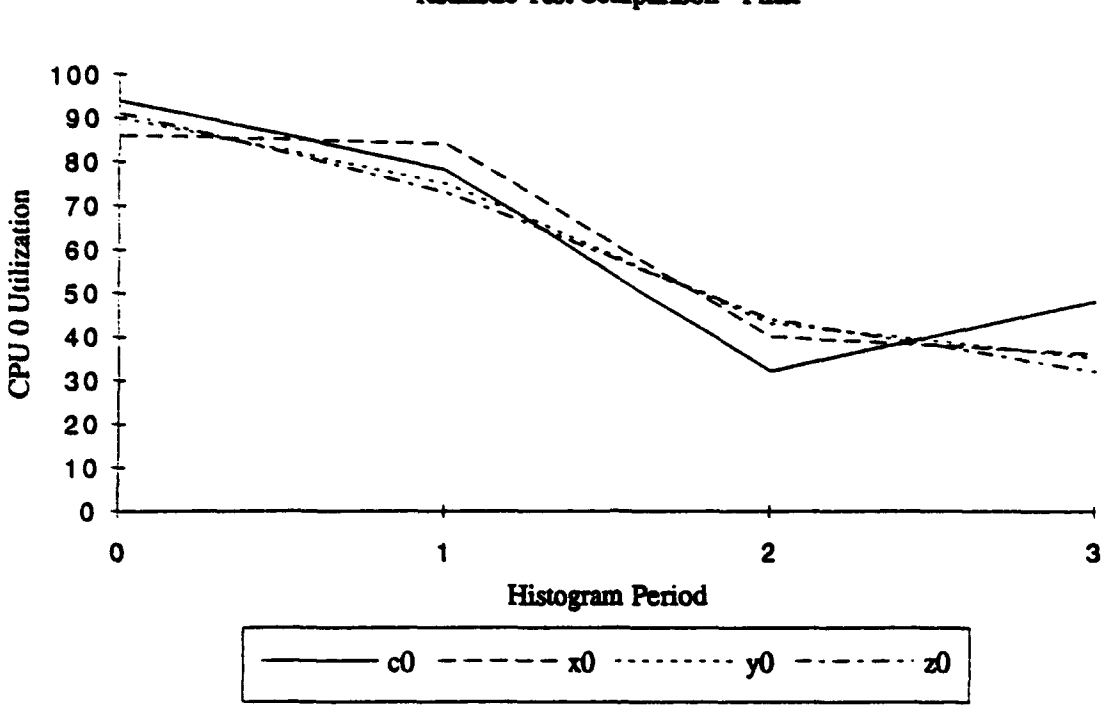


Figure 22 Stress Breakpoint Test--Collisions that Affect CPU 64

performance of the machine. In the first of this series of runs, each bus was assigned two CPUs, two shared memories, or eight local memories. The other two runs overlaid the two local memory bus sets over the lower half of one CPU bus set or the other. Refer to the last section of Chapter four for a detailed description of the test sets. The baseline run from the first suite of tests was also used as a baseline for this suite. Both suites used identical scenarios except for the configuration of the bus overloading.

The graphs in Figure 23 shows that for all three runs (and for the baseline run in the first suite, test case 'c'), the CPU utilization graphs follow the same basic curve pattern for each respective process. The matrix multiplication runs all start at around 35%, increase to 60% within the first 3 histogram periods and then remain near the 60% range for the rest of the run time. The significant drop for the matrix multiplication operation in test case y (line y1) was caused by this CPU waiting at the BARRIER for CPU 123 to finish its processing. CPU 64 was at the BARRIER for nearly half of the histogram period, while CPU 123 was running at 55% utilization at the time. The median filtering examples also seem to be consistent over the various loading scenarios. They all begin around 90% utilization and then continue to drop at a similar rate until they terminate near histogram period 4. This demonstrates that a careful overloading of the buses can be made without causing a significant degradation in the processing power of the system.

Figures 24 and 25 show the collisions on buses that could affect the performance of the particular CPUs. These are the buses that handle traffic to the CPU itself, buses handling traffic the CPU's local memory, and all buses handling traffic for the shared memories accessible to the CPU. It is interesting to note that, as anticipated in Chapter 4, the overloading of buses with multiple local memory assignments has the least impact in terms of collisions when compared with CPUs or shared memories for process 1 (matrix multiplication). This is not true, however, for the local memories associated with process 0. The percentage of collisions dramatically increased for this process (median filtering).



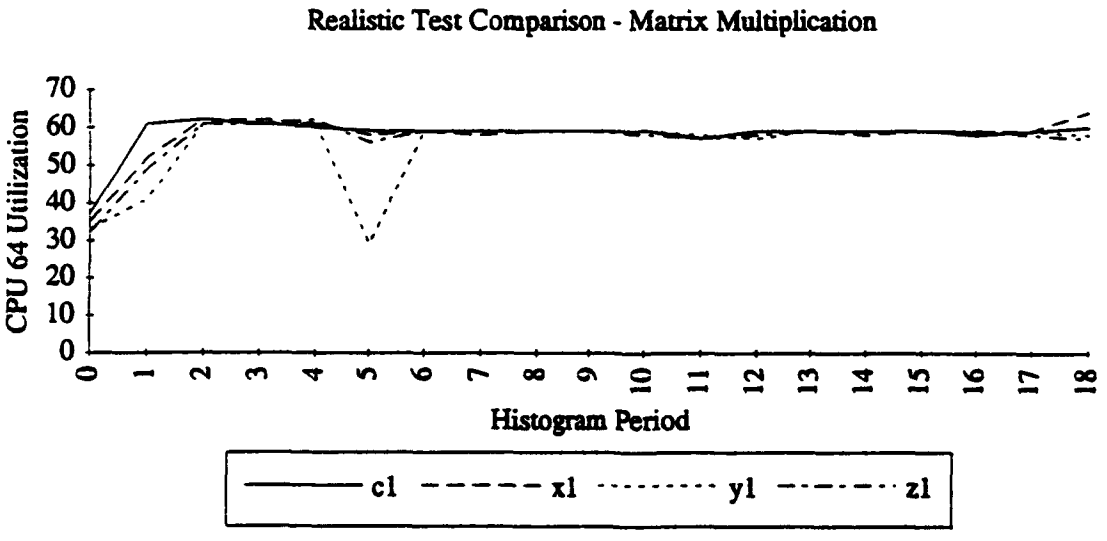


Figure 23 Realistic Test--Utilization for Relative CPU 0 for both processes

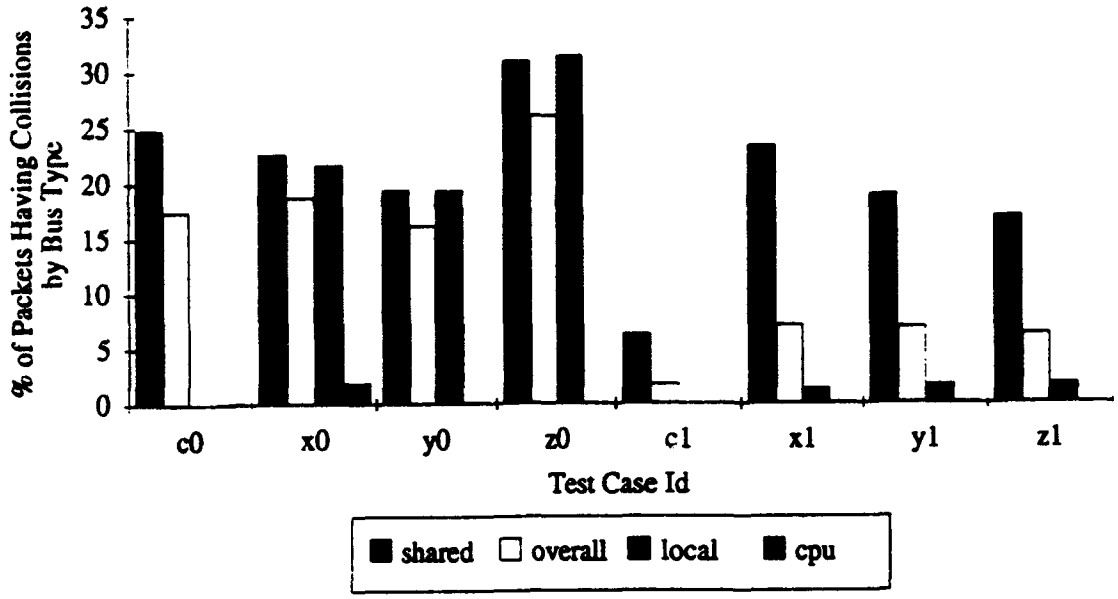


Figure 24 Realistic Test--Percentages of Packets having Collisions for Buses that Affect Relative CPU 0 for Both Processes

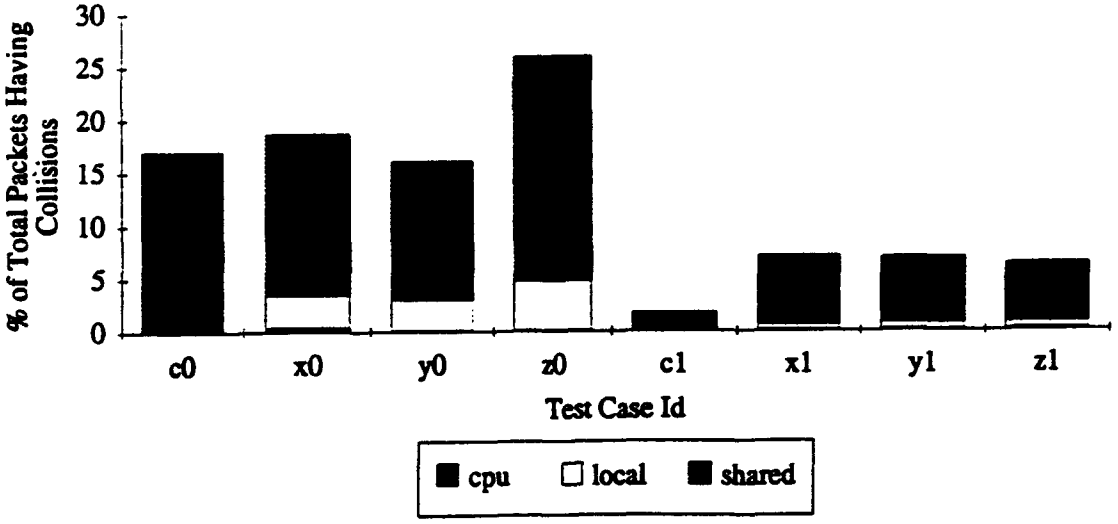


Figure 25 Realistic Test--Collision Percentages by Bus Type of Total Packet Counts for Buses that Affect Relative CPU 0 for Both Processes

However, Figure 25 shows that the local memory accesses were such a small fraction of the total memory accesses that there was a minimal change in the overall collision rate between case c0 and x0, y0 and z0.

5.3 **Results of Interleaving Comparison**

Following the execution of the two interleaving test cases described, the overall memory access patterns were examined to determine if a significant difference existed. Figure 26 is a graphical representation of the two access patterns. This graph depicts total number of access to the respective memory banks over the life of the process execution. The access pattern in the foreground is from the standard interleaving, with the one in the background representing the PBI accesses to the individual memory banks.

It can be seen that the standard interleaving pattern has a major spike at memory 35, with a secondary high mark for memories 10, 11, and 12. The PBI pattern also has a spike occurring at memory 60. The PBI pattern does not have any other significant peaks. The appearance of a more balanced access pattern (with the exception of the single peak) in the PBI run is supported by the comparison of the standard deviations of the two patterns. The PBI run had a standard deviation of 1007.18, while the standard interleaving run had 1319.8 for its standard deviation. These results show that the standard interleaving run had a less uniform distribution of accesses than the PBI.

This evaluation suggests that while the PBI did not completely solve the unbalanced memory access pattern seen with the standard interleaving, it did even the load somewhat. This further suggests that should a paradigm for more optimal access matrices be developed PBI could be used to evenly distribute the access patterns to a greater extent.

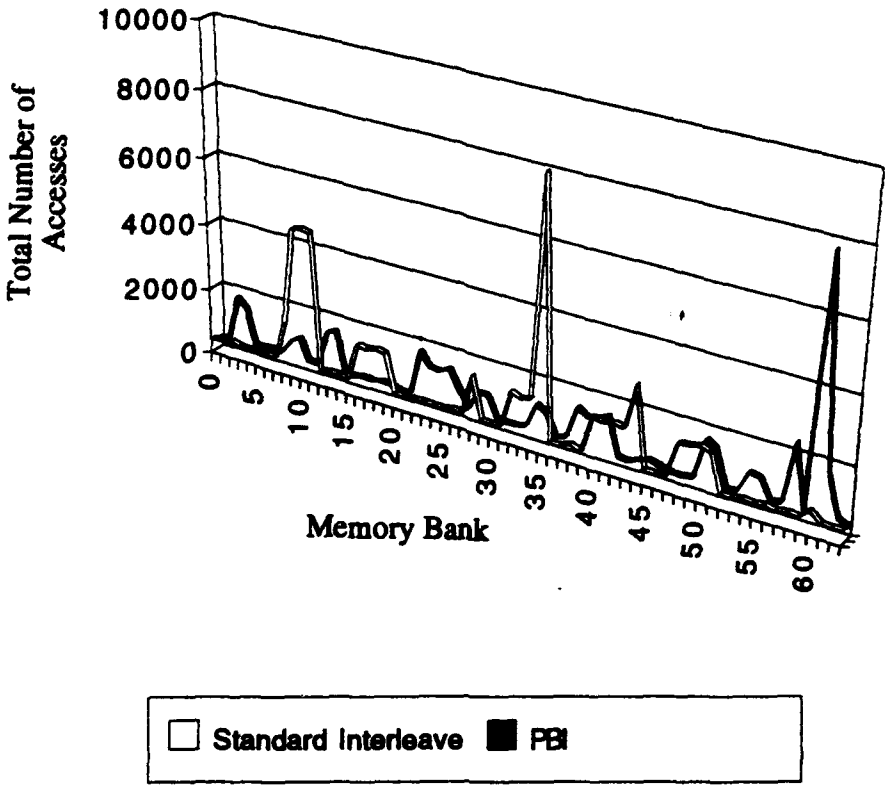


Figure 26 Comparison of Standard Interleaving Memory Access Pattern with Permutation Based Interleaving Memory Access Pattern

5.4 Conclusion

The circuit descriptions in Chapter 3 and simulation results in this chapter show that the PBI scheme can indeed be implemented. The results presented in this chapter have shown that it is feasible and desirable to utilize bus overloading within the MCA. This gives the flexibility to free more bus channels for use by other processes while having a minimal impact on CPU performance. A proof of concept test case, comparing memory access patterns for standard and permutation based interleaving, has shown that while there was not a dramatic difference between the two, there is potential for achieving a more evenly distributed memory access pattern.

VI Conclusions and Suggestions for Further Research

6.1 Conclusions

It has been shown that a permutation-based interleaving (PBI) scheme is possible even for architectures with a variable number of memory nodes such as the multiple channel architecture (MCA). It has further been demonstrated that the translation for this scheme can be done in hardware The algorithm associated with this interleaving scheme has been implemented in a multi-processor simulator and the results of some of the memory selections were verified by manual inspection of processor snapshots.

It has also been shown, through simulation results, that it is possible to overload singular buses with communications packets from multiple processes destined for more than one node. It was seen that indiscriminate overloading can have a serious degrading effect on CPU performance. It was also seen that a careful assignment of multiple nodes to single buses can be made that will have little or even no impact on the operational capability of the architecture.

6.2 Further Studies

A general rule for making bus overloading assignments was not developed. As with many situational analyses, the optimal assignment of nodes to buses may only be determinable on a case by case basis. This determination would need to take into account all factors involved such as numbers of processes, numbers of CPUs and memories, type of algorithms to be executed, etc. One possible area for further study would then be to see if some general heuristic could be developed to help users or operating systems determine the optimal, or at least a superior, mix of bus assignments.

In Chapter 3 it was mentioned that the line sizes for the local and shared caches need not be the same but will be fixed for the processor elements. At this writing, these line sizes have not been determined. A study is needed to determine what the line sizes will be for the respective caches.

This work did not involve an effort to ensure the coherency of the shared memory data Translation Lookaside Buffer (TLB). As explained before, the data for these simulation runs are all loaded into the simulated memory so no swapping of data to/from disk is necessary. This removes the need for a TLB, therefore one is not implemented in the simulator. Shared data TLB coherency could be a problem in an architecture where swapping of data is expected. Whenever shared data is brought into memory, all CPUs must be notified so that their respective TLBs can be updated. Similarly, they must know of shared data that is swapped out of memory. Another research effort should be initiated to solve this problem.

Additionally, for every change in the XOR matrix used in the PBI scheme, there is a possibility of a change in the permutation of the data subblocks within the memory banks. Sohi mentions in his article that certain matrices will give better results based upon the most prevalent strides in the algorithms used. He does not, however, give any answer as to how to match matrices with algorithms [Soh93]. Watching for his continued efforts may improve the choices of matrices for the MCA.

There has not been a decision made regarding the actual page size for the memories in the MCA. Because the subblocks within the pages will be a power of 2, any page size that is a power of 2 will suffice. Hennesey and Patterson remark that internal fragmentation is negligible for pages sizes between 2K bytes and 8K bytes in systems with megabytes of memory like the MCA. They further mention that page sizes greater than 32K bytes tend to cause problems including a serious effect on I/O bandwidth. [Hen90]

In summary, the MCA is a parallel architecture with several ground breaking technologies. It utilizes a state-of-the-art interleaving scheme to interleave data over arbitrary numbers of memory banks. This interleaving improves the balance of accesses to memory nodes. The MCA employs new tunable laser technologies to load bus channels with data destined for more than one processor, memory or I/O device. This overloading, when done with care, can be accomplished with minimal impact on CPU performance.

Appendix A

```
int get_memory_node ( Physical_Address,Log_2_Memory_Units, Transform_Matrix)

int Physical_Address;
int Log_2_Memory_Units;
char Transform_Matrix[LOG_2_MAX_MEMORY_UNITS][ADDRESS_BITS]


/* N O T E     This algorithm does not assume any units for Physical Address.
               In this simulator, the physical address will have been shifted right
               enough times appropriate to the cache-line size. (It has already been
               through the selective shifter)

{
    int Node_Number;
    int Node_Bit;
    char Physical_Address_String[ADDRESS_BITS];
    int a,b;

    Node_Number = 0;

    for ( a = ADDRESS_BITS-1;
          a >= 0;
          a--)
    {       /* transform the binary address into an ascii string of binary digits
               (adding 48 changes digit 1 to character '1' and digit 0 to character '0')*/
        Physical_Address_String[a] = ( Physical_Address % 2 ) + 48;
        Physical_Address = Physical_Address / 2;
    }

    for ( a = 0;
          a < Log_2_Memory_Units;
          a++ )
    {       /* perform bit-wise ANDing and parity computation  */
        Node_Bit = 0;
        for ( b = 0;
              b < ADDRESS_BITS;
              b++ )
        {       /* AND the two bits; add the result to Node_Bit;
                   Node_Bit must always be 0 or 1                          */
            Node_Bit = ( Node_Bit +
                         (Physical_Address_String[b] == '1' &&
                          Transform_Matrix[a][b]     == '1' )      ) % 2;
        }
        Node_Number = ( 2 * Node_Number ) + Node_Bit;
    }

    return ( Node_Number );
}
```

Bibliography

[Bal86] Balance Technical Summary: 2-4 to 2-6, A-7 to A-8, B-1 to B-9 (November 19, 1986)

[Edl85] Edler, Jan, and others. "Issues Related to MIMD Shared-Memory Computers: the NYU Ultracomputer Approach", IEEE Proceedings, 12th International Symposium on Computer Architecture: Boston, MA, 126-135 (June 1985)

[Den70] Denning, Peter J. "Virtual Memory", Computing Surveys, 2: 153-189 (September 1970)

[Kim91] Kim, Michelle Y. and Asser N. Tantawi. "Asynchronous Disk Interleaving: Approximating Access Delays", IEEE Transactions on Computers, 40: 801-810 (July 1991)

[Hen90] Hennessy, John L. and David A. Patterson. Computer Architecture: A Quantitative Approach. 437,484. San Mateo CA: Morgan Kaufmann Publishers, Inc., 1990

[Lin88] Linke, R. A. and A. H. Gnauck. "High capacity coherent lightwave systems", IEEE Journal of Lightwave Technology, 6: 1750-1769 (November 1988)

[Mil90] Milenkovic, Milan. "Microprocessor Memory Management Units", IEEE *Micro*: 70-85 (April 1990)

[Pfi85] Pfister, G. F., and others. "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture", Proceedings, 1985 International Conference on Parallel Processing: 764-771 (1985)

[Rei93] Reisner, John A. Design of a Shared Coherent Cache for a Multiple Channel Architecture. MS thesis, AFIT/GCS/ENG/93D-19. School of Engineering, Air Force Institute of Technology, Air University, Wright-Patterson Air Force Base OH.

[Soh93] Sohi, Gurindar Singh. "High-Bandwidth Interleaved Memories for Vector Processors--A Simulation Study", IEEE Transactions on Computers 42: 34-44 (January 1993)

[Wai92] Wailes, Tom S. Multiple Channel Architecture: A New Optical Interconnection Strategy for Massively Parallel Computers. Ph.D. dissertation. Purdue University, West Lafeyette, IN. August 1992 (TH397-36)

[War92] Warren, Karen. "PDDP: A Parallel Data Distribution Preprocessor", The 1992 MPCI Yearly Report: Harnessing the Killer Micros: 42-46 (August 1992)

Vita

Capt. John N Armitstead was born December 7, 1960 in Salt Lake City, Utah. He graduated from Jordan High School in 1979. After serving a two-year religious mission for the Church of Jesus Christ of Latter-Day Saints, he attended Utah State University, graduating in the fall of 1986. He received his Air Force Commission through Officer's Training School, Lackland AFB, TX, in the spring of 1987. He served as a software programmer and quality assurance analyst for the Air Force Global Weather Central, Offutt AFB, NE from 1987 until he was selected to attend the Air Force Institute of Technology as a master's degree candidate in 1992.

Permanent Address:

10667 Single Jack Circle
South Jordan, UT 84095

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE
VIRTUAL MEMORY MANAGEMENT AND VIRTUAL BUS OVERLOADING ON MULTIPLE CHANNEL ARCHITECTURES

5. FUNDING NUMBERS

6. AUTHOR(S)
John N Armitstead

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/93D-1

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

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Today's computational environment requires the processing capabilities available only through parallel architectures. The bottleneck that limits the potential of parallel processing is communication between processors, memories, and other hardware devices. A proposed multiple channel architecture (MCA) utilizes tunable semiconductor lasers and fiber optic cables that serve as the communication medium between processor, memory, and I/O nodes. A memory management unit (MMU) was completely described and implemented in a multiprocessor simulator. A permutation-based interleaving (PBI) scheme was utilized to reduce the chance of memory access collisions. Virtual bus utilization, number of collisions, and message traffic patterns were studied under various amounts of overloading. Results show that it is possible to maintain processor efficiency while reducing demand for channel availability.

14. SUBJECT TERMS	15. NUMBER OF PAGES
memory management, interleaving, permutation-based interleaving, PBI, tunable semiconductor laser, bus overloading, multiple channel architecture, MCA, parallel architecture	72
	16. PRICE CODE

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