Design and Analysis of a Memory Hierarchy for a Very High Performance Multiprocessor Configuration

by

Evan Michael Tick

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREES OF

BACHELOR OF SCIENCE

and

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January, 1982

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science, January 14, 1982

Certified by [illegible]
Arvind
MIT Thesis Supervisor

Certified by [illegible]
Tilak K. M. Agerwala
IBM Company Supervisor

Accepted by [illegible]
Alan C. Smith
Chairman, Departmental Graduate Committee

Design and Analysis of a Memory Hierarchy for a Very High Performance Multiprocessor Configuration

by

Evan Michael Tick

Submitted to the Department of Electrical Engineering
and Computer Science on January 14, 1982 in partial
fulfillment of the Requirements for the Degree of
Bachelor of Science and Master of Science in
Electrical Engineering

ABSTRACT

A memory hierarchy for a very high performance (one billion instructions per second) multiprocessor configuration is described. This hierarchy consists of two levels of private cache per hypothesized 128 MIPS processor, a shared memory and a switching buffer channel device. Unique problems introduced by the high performance multiprocessor environment are solved in the areas of mapping, replacement and consistency. A locking protocol and recovery mechanism are described for database system applications. A functional hardware description of the level two cache is given in HEX. An MP central-server queueing network model is derived for the system including processor/process affinity, write-back and both I/O and computation job classes each with differing block transfer sizes. Model solutions are analyzed.

Thesis Supervisor: Dr. Arvind
Title: Professor of Electrical Engineering

Acknowledgement

I am grateful to T. K. M. Agerwala for his assistence in helping solve many of the problems encountered in this work. I am also indebted to J. Cocke for the inception of the BIP MP system and his useful insights. The modeling effort derives from discussions with K. S. Trivedi, E. A. MacNair and C. H. Sauer. My understanding of locking and recovery was furthered by conversations with A. Chang. The **Speed Matching Buffer** organization is accredited to M. M. Denneau. I would finally like to thank D. DeGroot for his careful evaluation of this thesis.

Table of Contents

1. Introduction 9
    1.1 Statement of Problem 9
    1.2 Motivations 14
        1.2.1 Shared Memory 14
        1.2.2 Fully Associative L2 17
        1.2.3 I/O Bandwidth 19
    1.3 Literature Survey 24

2. Modeling 30
    2.1 MP Central Server Model 30
    2.2 Implementation 33
    2.3 Results 37

3. System Architecture 47

Memory Hierarchy

4. Memory Management 51
    4.1 L1 Mapping 56
        4.1.1 Conventional Map 59
        4.1.2 Hierarchical Map 64
    4.2 L2 Mapping 65
    4.3 L2 Page Fault Handler 75
        4.3.1 Replacement 78
            4.3.1.1 Algorithm 78
            4.3.1.2 Implemention 81
        4.3.2 Upward Transfer 85
            4.3.2.1 Orphans 85
            4.3.2.2 L2 Write-back 87
    4.4 Ms Mapping 91
        4.4.1 Conventional Map 92
        4.4.2 Proposed Map 98
    4.5 Ms Page Fault Handler 103
        4.5.1 Allocation Policy 109
        4.5.2 Replacement Algorithm 110

5. File Management 114
    5.1 SID Assignment 116
    5.2 Pathname to SID Conversion 116
    5.3 VA to BsA Conversion 117
        5.3.1 Segment Descriptor Table 117
        5.3.1 Backup Store Address Table 119
        5.3.2 Active Segment Table 120

6. I/O Management 127
    6.1 Input 127
        6.1.1 Demand Paging 127

6.1.2 Explicit Input 128
6.2 Burst Multiplexor Channel Device 128
6.2.1 Forward Path 129
6.2.1.1 Speed Matching Buffer 130
6.2.1.2 Burst Buffer 135
6.2.2 Reverse Path 141

7. Additional Points 143
7.1 Shared Memory Design 143
7.2 System Bus Design 146
7.3 File Adaptive Pre-Paging 146

Database Considerations

8. Lock Management 148
8.1 Locking Protocol 148
8.2 Implementation 151

9. Transaction Management 155
9.1 Commit 155
9.2 Abort 157

10. Recovery Management 159
10.1 Central Recovery 164
10.2 Local Recovery 166
10.3 System Log 167

11. Conclusions and Future Work 170

Appendix A: Technology 173

Appendix B: L2 Specification 176

Appendix C: File Manager Structures 200

References 204

Biographical Note 212

1. Introduction

1.1 Statement of Problem

The problem presented is the design of a memory system for a very high performance (aggregate billion instructions per second) multiprocessor configuration (BIP MP). There are five major goals in this design:

- develop a well balanced system, i.e. minimize the cost/performance ratio, aiming for a BIPS performance.
- don't push current disk or semiconductor memory technologies to achieve performance.
- develop a modular design which can be easily up or downgraded.
- support a multiprogrammed environment for both scientific and database applications:
    - scientific - efficient execution of relatively few jobs, each with large resource requirements.
    - database - efficient execution of many small transactions sharing very large databases.
        - guarantee level-3 consistency (GRAY78).
        - provide recovery mechanisms.
- develop an inexpensive yet accurate model of the system to assist in the design and to predict performance.

The BIP MP is comprised of eight hypothetical 128 MIPS processors, each with private memory, sharing a common primary memory (fig. 1.1). A large secondary memory, assumed to be an array of movable arm disks is required to support the large architected virtual address space. The I/O bandwidth generated by the attached processors creates an additional problem: the need for many *simultaneously active* disk

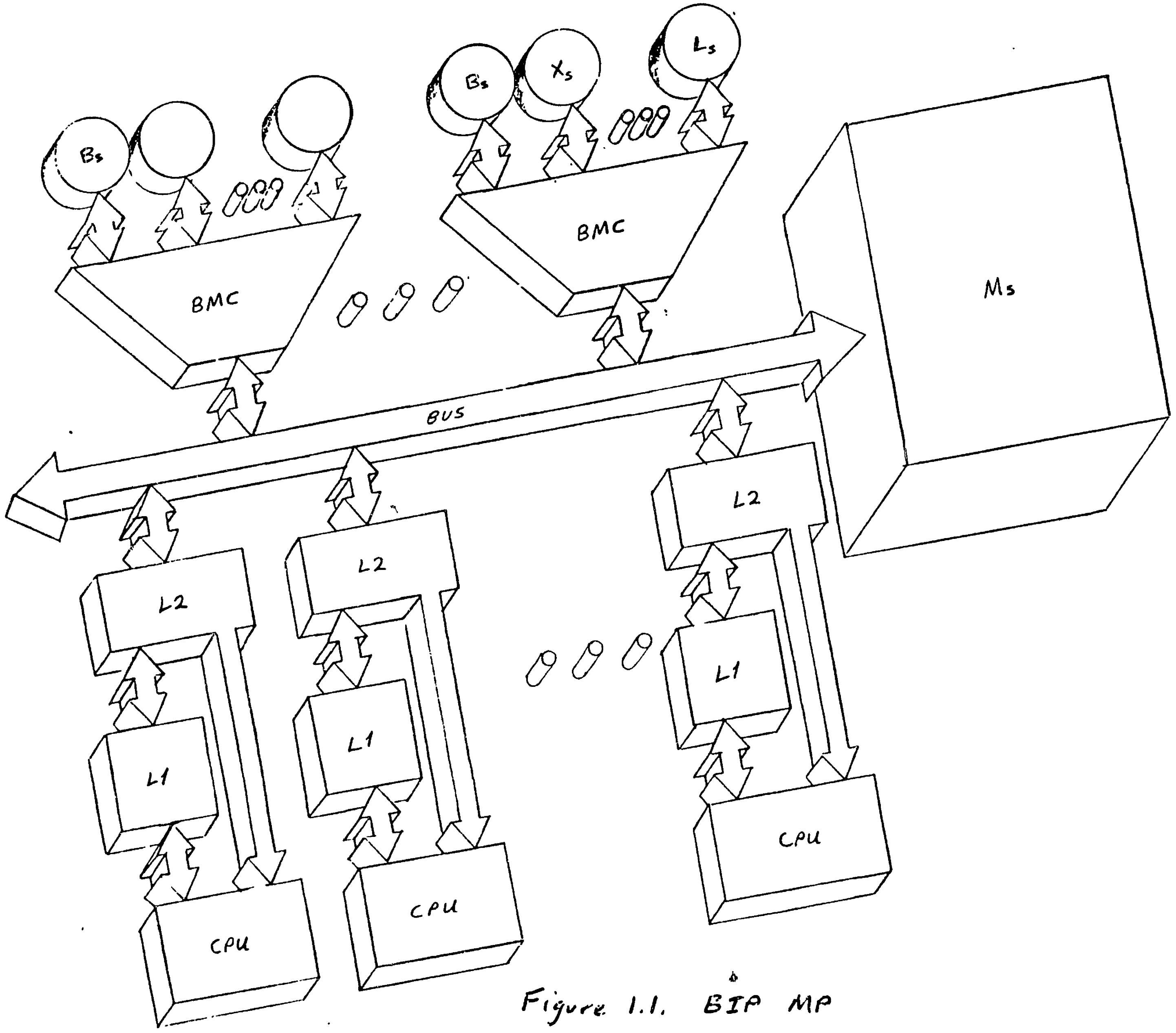


Figure 1.1. BIP MP

arms. The large shared memory is introduced to reduce this total I/O bandwidth requirement, i.e. to efficiently support the degree of multiprogramming necessary to achieve high performance.

The assumption of current disk technology implies a gross mismatch between secondary and primary memory bandwidths, 6 megabytes per second (MBS) and 1 gigabyte per second (GBS) respectively. A high speed channel device is proposed to make the datapath widths and rates compatible at this interface. There are four primary design goals for this channel device:

- the cost should be minimized to the extent that it is in proper proportion to the cost of the disks it services.
- the design must be modular to permit cost effective extension of secondary memory in reasonable units.
- the design must be fault tolerant to prevent channel hardware failure from crashing or bottlenecking the system.
- the design must be reliable and therefore simple.

To avoid a usually large switching penalty, no process [1] switch will be taken on a private memory miss, i.e. when the data object referenced by a processor is not resident in the associated private memory. This implies that the shared memory mapping must be efficient and more fundamentally that the private memories must be effectively managed.

Each private memory is equal in size to a small conventional main memory which suggests a similar implementation: a two level hierarchy. The level one cache, L1, is assumed to be a IBM 370/3033-like cache - a relatively small (32 - 64KB) high-speed set-associative memory which is transparent to the attached processor. The so-called

[1] process, job and transaction are used synonymously in this thesis.

level two cache, L2, is a much larger memory (1 - 4MB). There are two primary design criteria for L2:

- it must be fully associative to effect efficient page fault handling, i.e. reduce the private memory miss ratio by efficient replacement.
- it must be completely hardware controlled, i.e. transparent to the processor, to achieve a low L1 miss penalty.

The hardware design of a fully associative cache of conventional primary memory size is a unique problem.

Shared memory management is another complex problem that requires both a localized hardware controller and a dedicated high speed processor, the ***supervisor.*** The hardware controller must queue all shared memory requests from the processors and control data transfers to/from shared memory. Disk transfers are independently queued and controlled by the intelligent channel devices. The supervisor performs shared memory mapping (in hardware), page replacement (in software with hardware assist) and initializes all I/O to/from secondary memory (in software). It also pre-pages sequential files with hardware assist. The software functions are pieces of the ***global operating system*** running on the supervisor: the ***shared memory page fault handler*** and the ***I/O manager.*** Also required are the ***file manager*** and ***recovery manager.***

The file manager must deal with the issues of naming, i.e. translating a symbolic file name into a virtual address into a physical disk address. The large virtual space, orders of magnitude larger than those of conventional systems, creates uncommon problems keeping mapping structures and algorithms inexpensive in terms of both space and time.

The lock and recovery managers are additions to the global operating system forming the basis of a BIP MP database system. The lock manager must perform the setting and releasing of locks on data objects needed to guarantee *consistent* file manipulations. It is assumed in this thesis that the decisions of *when* and *what* to lock are made at compile time, leaving the problem of *how* to lock up to the manager at runtime. Again, the high performance goal necessitates developing a streamlined locking protocol and lock map structure for a very large file space. The complexity of a multiprogrammed multiprocessor multi-level memory hierarchy creates additional problems concerning the possible propagation of multiple copies of data objects and the subsequent ***interrogate problem.*** This problem is discussed throughout the thesis because it is infused in every level of the hierarchy.

The recovery manager must support deadlock recovery, volatile memory crash recovery and transaction failure recovery (non-volatile memory recovery, for instance from disk head crash, is not dealt with in this thesis). Again, the multiprocessor environment creates an interesting recovery problem. A local processor crash should not cause the other processors to crash or impact their performance. Therefore two types of memory crash recovery are needed: private and shared, corresponding to the respective memories. Private recovery must be transparent to the other processors.

Throughout the development of the BIP MP, the hierarchical design principle reappears again and again. The several abstract levels on which the system can be viewed are all hierarchies:

memory space
file space
memory maps
lock types
lock granularity
transaction log

1.2 Motivations

In this section several motivations for designing specific sections of the BIP MP are given. Arguments here are based on gross calculations, the purpose of which is to get a rough estimate of system requirements and bottlenecks in a very high performance environment. The underlying motivation for designing a memory hierarchy for the BIP MP is that scaling up the performance of a conventional system by about 100 (from 10 MIPS to 1 BIPS) scales up the necessary active disk arms by about a factor of 35 (20 to 700 arms), as will be shown. The hierarchy reduces this substantially.

1.2.1 Shared Memory

The introduction of the shared memory level between the fast multiprocessor private memories and slow secondary memory reduces the processor generated I/O bandwidth requirement. The reduction is by the factor of the shared memory's miss ratio. To justify the inclusion of an additional memory level, large enough to have a low miss ratio, it must be shown that the previous I/O bandwidth is intolerable.

To estimate the I/O bandwidth generated by a single processor, "Amdahl's constant" (AMDA70) may be used - assume one bit of I/O per executed instruction. This implies that a single 128 MIPS processor generates an I/O rate, R_{IO}, of

$$R_{IO} = \frac{128MbS}{8} = 16MBS$$

To calculate the number of simultaneously active disk arms necessary to support this bandwidth, the ***effective disk block transfer rate,*** R_{eff}, must be calculated. R_{eff} is the mean rate at which data is transferred from a single disk arm, taking into account both seek and latency.

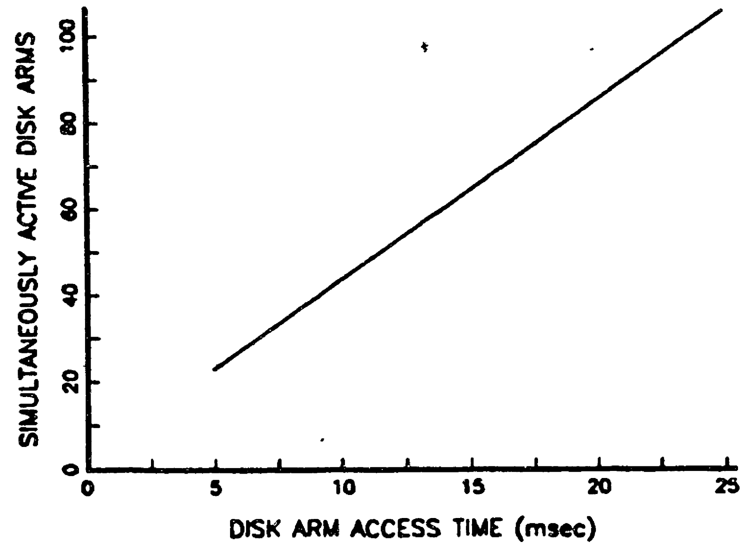
$$R_{eff} = \frac{b}{S + \frac{b}{R_{act}}}$$

where b is transfer size, S is the ***mean disk arm access time*** (seek plus latency) and R_{act} is transfer rate. Assume b = 4KB and R_{act} = 6MBS (Appendix A). The number of simultaneous active disk arms per processor is calculated as

$$arms = \frac{R_{IO}}{R_{eff}}.$$

This result is plotted in fig. 1.2. Taking S = 20ms as most realistic, this implies that the BIP MP requires 82 × 8 = 660 simultaneously active disk arms. It would be

FIGURE 1.2. DISK ARM ACCESS TIME VS. ARMS



difficult or impossible to build a bus capable of supporting that I/O bandwidth (over 4GBS).

Another way to view the shared memory's impact on system performance is to compare its function to those of conventional uniprocessor (UP) system components. In a conventional UP (fig. 1.3), large multiprogrammed virtual spaces are implemented by the primary memory ***demand paging*** to/from a secondary memory and a ***paging store.*** The secondary memory or backup store, Bs, is a non-volatile medium, usually disk, permanently housing the sum of the virtual spaces. This is the target of both ***implicit*** and ***explicit*** I/O. Explicit I/O is the loading of virtual data objects into primary memory through explicit file I/O commands. Implicit I/O is essentially demand paging to/from Bs, i.e. the loading of referenced virtual pages not resident in primary memory through default and the write-back of dirty pages. Most demand paging,

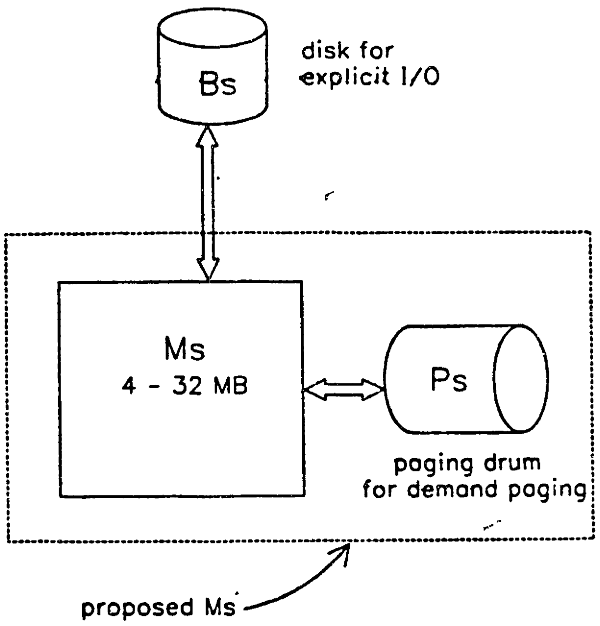


Figure 1.3. Conventional Main Memory

however, is to/from the paging store, Ps, usually a faster medium than disk, such as drum. Ps holds pieces of temporary offloaded working sets [1] belonging to switched out processes (working sets initially get into primary memory via I/O). The paging store is thus used to lower the switching penalty.

In the proposed system the shared memory is designed large enough (many times larger than the sum of the processor private memories) to obviate the need for a paging store. Actually this is not completely true. Implicit I/O will still take place, but to a lesser degree if all the working sets can fit in the shared memory comfortably. Of course one can always increase the degree of multiprogramming high enough to crowd the working sets, increasing the miss ratio and subsequently the amount of demand paging. This is offset against the increase in performance afforded by a large set of concurrently active processes to switch in during a process interrupt. Essentially this is the same tradeoff faced in a conventional system. An intelligent shared memory allocation policy must balance the degree of multiprogramming with the miss ratio (section 4.5.1.1).

1.2.2 Fully Associative L2

It was stated that L2 is fully associative to reduce the private memory miss ratio, essential for attaining the desired performance. A rough calculation is made of the expected degradation in a single processor's performance for various miss ratios. These numbers indicate that the miss ratio afforded by a fully associative memory is necessary.

[1] The term "working set" is used loosely in this thesis and is only superficially related to the standard definition (DENG80). The meaning here is closer to ***primary memory allocation,*** PMA, discussed in (BAER80).

Assuming a shared to private memory transfer size of 1KB, the ***mean shared memory access service time***, s, is estimated as

$$\begin{aligned} s &= latency + transfer \\ &= 1us + 1us = 2\ us. \end{aligned}$$

The transfer time assumes a 1GBS bus. The latency time assumes a 400ns chip access time, conservatively giving a 1us package access time. The mapping time is not significant (section 4.4.2).

Assume the private memory misses one out of every N instructions executed. Thus the mean aggregate and single processor ***inter-fault interval***, IFI_{mp} and IFI_{up}, are

$$IFI_{mp} = \frac{N\ instr}{1\ BIPS} = N\ ns$$

$$IFI_{up} = \frac{N\ instr}{128\ MIPS} = 8N\ ns.$$

The miss penalty, w, the total time spent waiting for a private memory miss to be serviced, can be estimated for large N by (DRAK67)

$$w = \frac{s}{1-\frac{s}{IFI_{mp}}} = \frac{2\ us}{1-\frac{2\ us}{N\ ns}} = \frac{2N}{N-2000}\ us.$$

This queueing formula takes into account contention between the processors for service. Performance can be measured as the mean rate at which private memory faults are serviced

$$perf = \frac{1}{IFI_{up} + w}.$$

The best attainable performance is therefore IFI_{up}^{-1} because the miss penalty approaches zero for small service times. Single processor performance degradation, equivalent to system degradation, d, where $0 < d < 1$, is thus defined

$$d = \frac{IFI_{up}^{-1} - \frac{1}{IRI_{up} + w}}{IFI_{up}^{-1}} = \frac{w}{IRI_{up} + w}.$$

It is desirable to keep degradation below a given value, D_{max}, thus

$$D_{max} > \frac{w}{IFI_{up} + w}$$

Substituting for w and IFI_{up} and solving for N,

$$N > 1750 + \frac{250}{D_{max}}.$$

This is also calculated assuming a 4KB transfer size, using

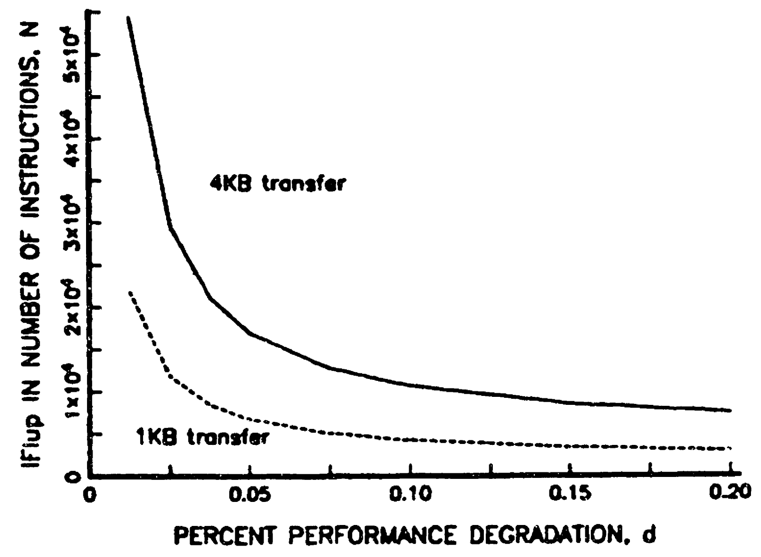
$$N > 4375 + \frac{625}{D_{max}}.$$

The non-linear relationship between performance degradation and private processor miss ratio is shown in fig. 1.4. To achieve low penalty, N must be large, i.e. hit ratio must be high. For conventional fully associative primary memories, a standard value for N is 10,000 (COCK81). A fully associative L2 is proposed to guarantee a low miss ratio.

1.2.3 I/O Bandwidth

To get an estimate for the degree of multiprogramming and I/O bandwidth, three simple application programs are analyzed. Assumed throughout are a disk arm access time of 20ms, eight processors (nop = 8) and explicit I/O capability for 64KB and 32KB block transfers.

FIGURE 1.4. L2 MISS PENALTY



Matrix Multiply

Matrix multiply is an $O(N^{1.5})$ operation where N is the number of elements in the matrix. A typical inner loop for a floating point (64 bits per element) matrix multiply, in 370/Assembler (IBM81), looks like:

```
LOOP  LD    0,0(6,4)       R4 = ptr to matrix A
                           (column major order)
                           R0 = single term product
      MD    0,0(6,5)       R5 = ptr to matrix B
                           (row major order)
                           do A(i,j)*B(j,k)
                           R6 = j
      ADR   1,0            sum over the row
                           R1 = partial sum
      BXLE  6,2,LOOP       increment j
                           R2 = 8 (1 DW/entry)
                           R3 = length of row
```

The following figures are of interest:

$N = 1024 \times 1024$	$= 2^{20}$
total I/O traffic : $S = 3 \times 1024 \times 1024 \times 8$	= 24 MB
R_{eff} = 64KB / (20ms + 64KB / 6MBS)	= 2.2 MBS
total I/O time : $t_{IO} = S / R_{eff}$	= 11 sec
total number of instructions : $I = 4 \times N^{1.5}$	$= 4.3 \times 10^9$
total processor time : t_p = I / 128 MIPS	= 34 sec
degree of multiprogramming : $dmp = t_{IO} / t_p$	= 1
simultaneously active arms : arm = nop x dmp	= 8
total I/O bandwidth : iob = arm x R_{eff}	= 18 MBS

The degree of multiprogramming is estimated as the ratio of the I/O time to the processor time because in the asymptotic limit with no switching penalty, processor and I/O device operation will completely overlap. Now assume the matrix is only 32KB or 1/256 the size of the above matrix. The matrix must be transferred from disk with a smaller block size of b = 32KB resulting in a lower R_{eff}. The following shows the drastic effect the smaller data object size has on I/O requirements:

$N = 64 \times 64$	$= 2^{12}$
R_{eff} = 32KB / (20ms + 32KB / 6MBS)	= 1.3 MBS
total I/O traffic : $S = 3 \times 64 \times 64 \times 8$	= 98 KB
total I/O time : tio = S / R_{eff}	= 74 ms
total number of instructions : $I = 4 \times N^{1.5}$	$= 1.0 \times 10^6$
total processor time : t_p = I / 128 MIPS	= 8 ms
degree of multiprogramming : $dmp = t_{IO} / t_p$	= 10
simultaneously active arms : arm = nop x dmp	= 80
total I/O bandwidth : iob = arm x R_{eff}	= 104 MBS

Sorting

Radix-exchange sort is an O(NlogN) operation where N is the number of entries in the list (KNUT73). Consider sorting a list of entries consisting of a single word (4B) key and an information field of M words. A simplified inner loop of the sort looks like (the simple method of information field transfer presented here could be alleviated with pointers):

```
LOOP1    L      5,0(1,3)            R3 = ptr to list K
                                   R1 = i (top index)
                                   R5 = K(i)
         SLA    5,BIT(2)           BIT(2) = search bit #
         BC     NEG,SKIP
         BXLE   1,6,LOOP1           R6 = 1
                                   R7 = j (bottom index)
         BC     UNC,BOT            finished partition
LOOP2    L      5,4*(1+M)(7,3)      R5 = K(j+1)
         SLA    5,BIT(2)
         BC     POS,SWITCH
SKIP     BXH    7,0,LOOP2           R0 = -1
         BC     UNC,BOT            finished partition
SWITCH   L      5,4*(1+M)(7,3)      first switch keys of
         L      4,0(1,3)            K(i) and K(j+1)
         ST     4,4*(1+M)(7,3)
         ST     5,0(1,3)
         L      5,4*(2+M)(7,3)      first switch keys of
         L      4,4(1,3)            K(i) and K(j+1)
         ST     4,4*(2+M)(7,3)
         ST     5,4(1,3)
            .
            .
            .
         L      5,4*(1+2*M)(7,3)
         L      4,4*M(1,3)
         ST     4,4*(1+2*M)(7,3)
         ST     5,4*M(1,3)
         BXLE   1,6,LOOP1
BOT
```

The following figures are of interest:

N	$= 2^{20}$
M	= 7
R_{eff} = 64KB / (20ms + 64KB / 6MBS)	= 2.2 MBS
total I/O traffic : S = 2 x 32 x 2^{20}	= 64 MB
total I/O time : $t_{IO} = S / R_{eff}$	= 29 sec
total number of instructions : I = 43 x NlogN	= 8.0 x 10^8
total processor time : t_p = I / 128 MIPS	= 6.3 sec
degree of multiprogramming : dmp = t_{IO} / t_p	= 5
simultaneously active arms : arm = nop x dmp	= 40
total I/O bandwidth : iob = arm x R_{eff}	= 88 MBS

The following shows beneficial effects of reducing data object size.

N	$= 2^{16}$
M	= 1
R_{eff} = 64KB / (20ms + 64KB / 6MBS)	= 2.2MBS
total I/O traffic : S = 2 x 8 x 2^{16}	= 1MB
total I/O time : tio = S / R_{eff}	= 450 ms
total number of instructions : I = 38 x NlogN	= 4.0 x 10^7
total processor time : t_p = I / 128 MIPS	= 310 ms
degree of multiprogramming : dmp = t_{IO} / t_p	= 2
simultaneously active arms : arm = nop x dmp	= 16
total I/O bandwidth : iob = arm x R_{eff}	= 35MBS

Matrix Addition

Matrix addition is an O(N) operation where N is the number of elements in the matrix. A simplified inner loop (floating point elements) looks like:

```
LOOP  LD    0,0(6,1)      R1 = ptr to matrix A
                          (row major order)
      AD    0,0(6,4)      R4 = ptr to matrix B
                          (row major order)
                          do A(i,j) + B(i,j)
                          R6 = j
      STD   0,0(6,5)      R5 = ptr to matrix C
      BXLE  6,2,LOOP      increment j
                          R2 = 8 (1 DW/entry)
                          R3 = length of row
```

The following numbers show that the operation is I/O bound.

N = 1024 x 1024	$= 2^{20}$
total I/O traffic : S = 2 x 1024 x 1024 x 8	= 17 MB
R_{eff} = 64KB / (20ms + 64KB / 6MBS)	= 2.2 MBS
total I/O time : tio = S / R_{eff}	= 7.7 sec
total number of instructions : I = 4 x N	= 4.2 x 10^6
total processor time : t_p = I / 128 MIPS	= 33 ms
degree of multiprogramming : dmp = t_{IO} / t_p	= 233
simultaneously active arms : arm = nop x dmp	= 1860
total I/O bandwidth : iob = arm x R_{eff}	= 4.1 GBS

1.3 Literature Survey

Many of the concepts presented in this thesis stem from a collection of systems. This naturally follows from design goals combining high performance uniprocessors such as CRAY-1 (SITE78) together in a multiprocessor configuration with a clean and useful system architecture, such as MULTICS (DALE68), System R (CHAM81) and System/370 MP (MACK74). Although individual concepts may be commonplace, their collection in a high performance implementation is unique.

Memory Hierarchy

(SMIT78a) outlines current research topics concerning memory hierarchy design. He sites cache mapping and consistency in an MP environment as two significant unsolved problems. No mention is made of a level two cache however. An additional problem outlined is the user view of the hierarchy, i.e. the extent to which the hierarchy must be architected to achieve performance. These issues are addressed in this thesis.

(KATZ71, PUGH71, MEAD70) present early discussions of memory hierarchy design issues. Only simple mapping and replacement overviews are presented. Many papers discuss level one cache organizations and mapping, replacement and update policies, for instance (LIPT68, POHM73, KAP73, BELL74, POHM75b, RAO78, SMIT78b). No evaluations of hash-chain table map organizations are compared. The "inverted" page table mapping technique used in System/38 (HOUD81) is perfectly suited to an arbitrarily large name space and is used in the BIP MP. (AGRA77) uses simple analysis to evaluate a system similar to the BIP MP without the L2 level.

(DESA80, TANG76) suggest a level two *system* cache shared by the multiprocessors. This organization is valid, however it is susceptible to system failure when the shared cache crashes. In addition, the mapping problems of a large shared cache are more difficult than those encountered in a smaller local second level cache. (SPAR78) suggests a level two *private* cache, but stipulates that it be set-associative, store-through and real addressed. In other words the cache holds a partial image of the shared memory. The ***cross-interrogate problem*** is still there, although somewhat lessened by L1 store-in and L2 store-through policy. The ***synonym problem*** (section 4.1.1) is still there also. These problems are avoided in the BIP MP.

(LANG77) suggests that for a uniprocessor, a separate "database buffer" in faster-than-main-memory technology be used for demand paging in a database environment. The results presented there are for small databases and it is doubtful that performance scales linearly with main memory size. In essence, the proposed shared memory is large enough to assume this function.

File and I/O Subsystems

The BIP MP virtual space is substantially larger than those of standard machine architectures, e.g. System/370 (IBM81). MULTICS, with 64 pages per segment (BENS72) avoids many of the problems faced here with 8K pages per segment. Conventional mapping, replacement and file management schemes become inefficient and intractable in the proposed system.

File manager demands in a high performance environment are summarized for CRAY-1 DEMOS in (POWE77). The I/O bandwidths stated there, roughly 10MBS/processor, are equivalent to the BIP MP requirements. Interesting data is given on expected I/O transfer sizes in a scientific environment. A simplistic A/B channel buffering mechanism is described which is not effective in a multiprogrammed environment (all results presented assumed pure sequential accesses). Elaborate "preallocation" and "read-ahead" disk strategies are suggested which are facilitated by the small virtual space. (SITE78) analyzes the CRAY-1 memory hierarchy indicating the inefficiencies of the I/O subsystem.

The Eclipse M/600 channel device running at 10MBS for a virtual space of 6GB (COMP78), is similar to performance capability necessary in the proposed devices.

(TOKU80) discusses "file adaptive" techniques, in the context of the IDC, a 16MB ***disk cache*** of the ACOS system. Sequential files are prefetched and replaced in a non-LRU manner in the IDC, and temporary files are not transferred back to disk. These general techniques have been outlined for shared memory replacement in the BIP MP. (BRAN81b) suggests various schemes for increasing I/O performance, such as multiple data paths among the I/O devices serviced by a single controller, a track buffer per spindle and a full cache servicing several disks.

Database Systems

(GRAY78) describes the general two-phase locking protocol and two-phase commit for a *distributed* system in addition to some possible solutions to the ***convoy problem.*** The general locking protocol in (GRAY78) obeys the ***rising-falling rule*** and requires a large log with both ***redo*** and ***undo*** records and a complex recovery procedure. (NAUM79) mentions a less costly but less general scheme - releasing exclusive locks at transaction commit. This is the basis of the work presented here.

The commit protocol described in (GRAY78) is needed to synchronize the distributed transactions into all committing or all aborting - the "general's paradox". This problem is not encountered in the proposed system because transactions are assigned to one and only one processor. The COMMIT procedure described in this thesis is termed *two-phase* to distinguish when a transaction becomes recoverable.

An example of a recent database system is System R. (CHAM81) gives an overview of System R's success at locking and recovery. Three important results are noted.

- Level-3 consistency was the only decidedly necessary locking level.
- The disk "shadow page" scheme impacted performance too much. (HARD79) describe one method of implementing "shadow pages".
- Convoys were avoided by giving running transactions priority for receiving locks on high traffic resources.

(RIES77, RIES79, POTI80) use queueing network models to analyze the effect of lock granularity on lock contention. These models are similar in that they assume transaction reference targets can be described by a uniform distribution over the database. (RIES79) suggests a lock hierarchy using large lock granularity for transactions touching greater than 1% of the database and transactions doing sequential accesses. The basic conclusions reached are that optimal granularity is application dependent and for a "claim (locks) as needed" policy with "mixed sized transactions", a "lock hierarchy with fine granularity" is suggested. These proposals were considered in the BIP MP design.

Modeling

F--ly modeling of memory hierarchies began in the context of optimization techniques. (RAMA70) presents an early model for cost/performance minimization of a multiprogrammed general memory hierarchy making the assumption of random reference targets. Integer programming solutions of modular memory design optimization are also presented. Stack algorithmic analysis, introduced in (GECS70), began a series of efforts aimed at reference trace driven models. The advantage of this modeling method is its speed. A disadvantage is that most implementable replacement algorithms are only approximations to stack algorithms. Another problem is determining "typical" workloads to trace. (MATT71, LIN72) use stack analysis for evaluating two and three level hierarchies. (MACD75) presents a stack algorithmic technique for minimizing either mean access time or cost/performance in a general memory hierarchy. The scheme assumes pure LRU replacement at each level in the hierarchy, however.

Geometric programming optimization techniques were further developed in (CHOW74, CHOW76, WELC76). Mean memory access time was minimized at a given cost constraint for general memory hierarchies. These studies describe memory performance and cost as power functions and deal with uniprogrammed UP configurations only. More recently, Lagrangian multiplier optimization has been used on queueing network models of memory hierarchies (CHAN78, TRIV80, TRIV81). These studies deal with multiprogrammed UP configurations.

The view taken here is that an accurate model which is not restricted to analytic optimization techniques is more useful. Optimization over the model's parameter space is facilitated if the model is inexpensive to solve. (GECS74, POHM75a, REGE76) describe early cyclic queueing models of memory hierarchies. (GECS74) describes an optimization technique for a multiprogrammed UP general hierarchy. (POHM75a, REGE76) use simple queueing models to evaluate the cost/performance of an electronic (as opposed to magnetic) ***paging store.*** (YEN81) gives extensive queueing network models of a system similar to the BIP MP without an L2 level. That study concerned level one cache management policy effectiveness. Other examples of recent queueing network models for multiprogrammed UP configurations are found in (SAUE81, HEID81, TRIV81). The multiprogrammed MP model presented in this thesis is based on these efforts. Implementation in RESQ2 (SAUE80a, SAUE77) permits innovative modifications, namely the accurate modeling of differing bus transfer sizes, write-back, process/processor affinity, and bus contention.

2. Modeling

A queueing network model of the BIP MP was developed. The model assumes exponential distributions for memory reference arrivals and both computation and I/O service times. This model is used to justify and refine the rough calculations made in section 1.2. It is used primarily as a design aid to accurately measure system bottlenecks. The model described is, as are all models, only an approximation of reality, but it does give confident *relative* results over its parameter space.

2.1 MP Central Server Model

The model used is a multiprocessor equivalent of the **central server model** (ALLE80, SAUE80b, SAUE81). The model (fig. 2.1) is comprised of a single chain closed network. Job population is simply the degree of multiprogramming of the system. Jobs are not homogenous. There are two job ***classes:*** computation and I/O jobs. Within each class are m subclasses, one for each work processor. Servers P_1 - P_m are the 128 MIP work processors, P_0 is the supervisor processor, D_0 - D_{n-1} are I/O devices (including channel devices) and BUS is the system bus.

One can conceptualize the model by visualizing the jobs as tokens flowing counterclockwise around the network. Each token is either in service at or waiting for (enqueued at) a resource. When released from a server (job releases resource), a job often finds itself faced with more than one possible ***branch*** or path to take. Associated with each branch is a ***branch probability***, either explicitly marked, as in the case of the shared memory hit/miss ratio, or implicitly uniform over the choices, as in the case of the I/O devices. Thus a job leaving a processor, after a mean service time equal to

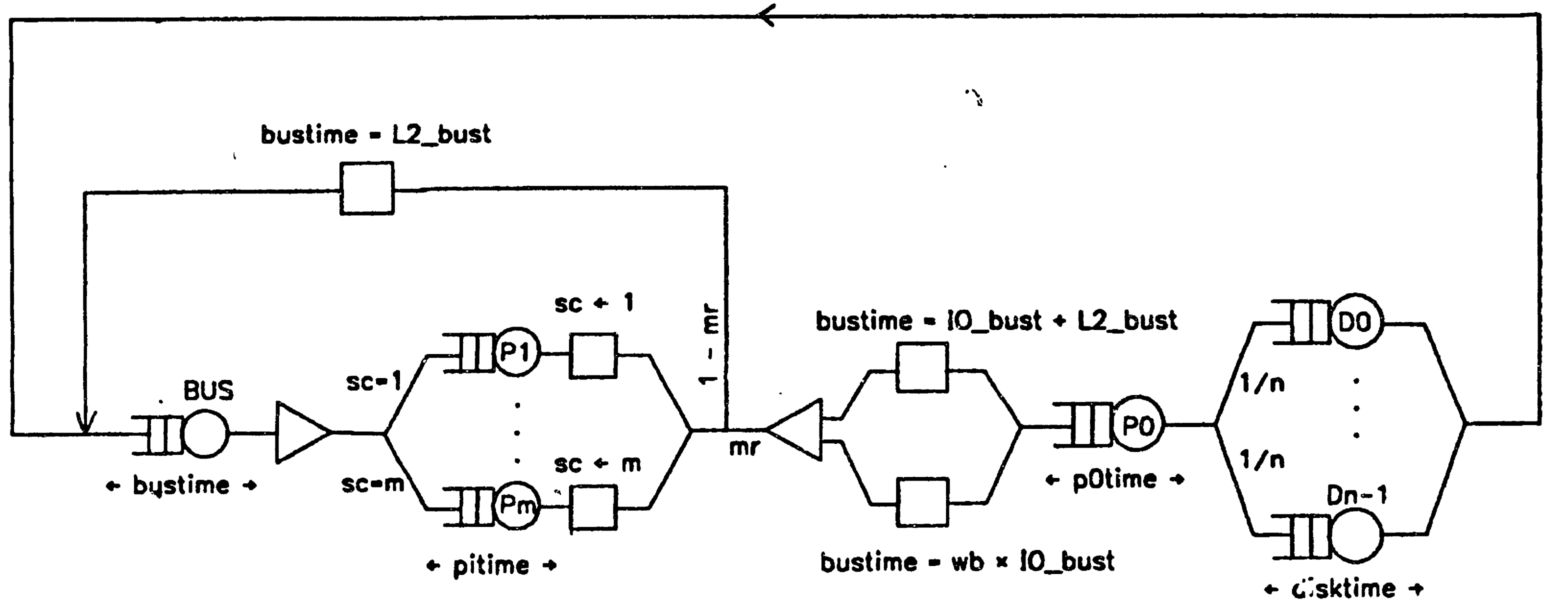


Figure 2.1. MP Central Server Model

IFI_{up}, will access shared memory and either continue processing if it hits, or perform I/O if it misses. A hit in shared memory simply involves a bus transfer to the associated private memory, hence the inner feedback path. The I/O taking the outer loop is modeled in three stages:

- P_0 - The supervisor writes a channel control program for the job, initiating the I/O.
- D_i - The intelligent channel device, once granted the proper disk, executes the channel program, transferring the requested data object from disk to channel buffer.
- BUS - The channel device is granted the bus and bursts the data object to the shared memory.

The job subclasses are introduced to route that portion of the multiprogrammed set belonging to a processor to that processor *only*. This models the normal consequence of shared memory references and also process/processor affinity imposed by consistency issues (section 8.0). The two job classes are introduced to permit different block sizes for I/O transfers and private memory transfers (different BUS service times).

Write-back of modified shared memory pages is modeled by creating an additional "write-back" token for every private and shared memory reference. This child token travels the same path as its parent, contending for the same resources and finally being destroyed after it completes its bus transfer. This does not model database applications accurately because in the database system all write-back is at transaction commit. If the transactions are short lived the current model is a good approximation. Note the write-back token is *always* created, modeling worst case; however, the write-back transfer size is a fraction of the request transfer size. An alternative approach would be to avoid write-back with a certain probability distribution. (SMIT79) analyzed applications where the percentage of total replaced pages which required write-back, i.e. were dirty, ranged from 20 - 60%. The distribution was irregular and application dependent. It is felt that the method used in this thesis is realistic or at least pessimistic because of the low probability of a large 64KB page remaining totally clean.

The central server model, as opposed to the cyclic queue model (SAUE81, SAUE80b, REGE76), defines each I/O device with its own queue. This is thought to be a more accurate description of the intelligent channel device queues than the multi-server view of I/O (one queue for all disks).

All jobs are given equal priority when contending for the bus. This is an accurate description because of the independent nature of the channel devices. In general, the intelligent channel devices and processors contend for the bus independently. Other systems may specify synchronization mechanisms with which to prioritize these bus requests, that is not the assumption here.

The number of disk arms in the model is actually *only half* the required number because of the channel device design (section 6.3). Essentially the channel devices have an effective blocking ratio of 0.5, i.e. only one of two attached disk arms can be active at any one time. More extensive channel device models were not considered necessary. The model presented incorporates all the I/O service time factors outlined in (BRAN81b) except for "missed reconnection delay".

2.2 Implementation

The model was implemented in RESQ2, the *IBM Research Queueing Package, Version 2* (SAUE80a), using the APLOMB simulation method of solution (SAUE77). The original models developed were kept simple to permit analytic solution techniques.

However several key aspects of the system could not be easily described. The answer was to use simulation techniques and pay the increased solution cost. The simulation model was solved on an IBM 370/3033 with 90% confidence limits of less than 10% mean relative error for resource utilizations. Confidence intervals were generated using the method of replications. A replication was defined as 33,000 simulation events or queue arrivals. Regenerative methods (SAUE81) could not be used because the model was too large. Transient effects of initial conditions were eliminated by discarding the first 10% of every simulation run. The initial condition was to place an equal number of jobs in each processor queue.

The job classes and subclasses were implemented in APLOMB using ***job variables*** and ***set nodes***, present as squares in figure 2.1. Each job's class and subclass are kept in two private tags or job variables belonging to that job. The subclasses are assigned at the set nodes associated with each processor. Once assigned they never change. The class tags are changed from computation to I/O class and back again at the SIO and L2 set nodes, respectively.

It was determined that L2 to Ms write-back has no first order effects in the model and so only I/O write-back jobs are created, speeding up the simulation. The write-back tokens are created by a ***fission node*** and destroyed at a ***fusion node***, present as triangles in figure 2.1. These children are conceptually in the same class as their parents, however they are given different tags at the SIOWB set node. This is done because the BUS service time is determined by the job class tag:

job class	**bus service time**
I/O	IO__BUST + L2__BUST
I/O write-back	WB x IO__BUST
computation	L2__BUST

where WB is the expected percentage of dirty lines in a page. The branch probabilities associated with the processors are determined by job subclass. The processors are modeled with the ***processor sharing discipline,*** PS (SAUE81). This essentially grants an infinitesimal time slice in round-robin fashion to each job enqueued at the processor.

The parameter space of the model was designed to test the limits of the system. Unfortunately, as is often the case with queueing network models, the space was in part constrained by the feasibility and cost of solving the model. The model was embedded in a PL/I control program used to feed it different sets of parameters. All service time distributions used, except for the BUS which has a fixed service time, are exponential, i.e. arrivals are Poisson with unity coefficient of variation. Model parameters and their ranges are as follows.

- nop : number of processors

  Eight processors are proposed for the BIP MP.

- Pitime : mean processor service time

  A single 128 MIPS processor missing in its private memory once every N = 10,000 executed instructions, has Pitime = IFI_{up} = 80us.

- P0time : mean supervisor service time

  The supervisor initiates I/O, i.e. it executes a conventional Start I/O instruction (IBM81), translating into roughly 1000 - 20,000 instructions (COCK81). Assuming 10,000 instructions and a 128 MIPS supervisor, P0time = 80us. This figure may be conservative, however the additional delay may be conceptualized as the effect of higher priority system jobs contending for the supervisor.

- disktime : mean effective disk transfer time

  Assuming a transfer size b, a disk arm access time S, and a rotational transfer rate R_{act} = 6MBS,

  $$disktime = S + \frac{b}{R_{act}}.$$

  Because disk arm access time is highly application dependent, an upper and lower bound are modeled: S = 20ms and 8ms. (JONE81a) gives similar delays for the IBM 370/168. The 8ms corresponds to no seek and mean latency. The 20ms corresponds to either no seek and full latency - the "missed revolution" effect (JONE81a), or mean seek and mean latency. This agrees with (BRAN81b) who uses 11.2ms as mean seek time, assuming 30% of all I/O requests require no seek.

Two block transfer sizes are also modeled: b = 4KB and 64KB.

- L2__bust : mean bus service time for L2 transfers

  Only 4KB transfers are modeled. The system bus is designed for transfer rates of up to 1GBS; however, this is admittedly optimistic. A lower bound of 512MBS is also modeled for comparison.

- IO__bust : mean bus service time for disk transfers

  Both 4KB and 64KB disk transfers are modeled for both bus speeds.

- wb : percentage of dirty lines in page

  wb = 0.3 was modeled, estimated from an assumed 2:1 load to store ratio. The write-back factor can be approximated this way because I/O transfer size granularity is relatively sm^ll - 1KB.

- nod = (64,128,256) : number of disks

- dmp = (70,105,140,210) : degree of multiprogramming

- mr = (0.05,0.075,0.1,0.2) : shared memory miss ratio

The parameter sets are summarized below. Sets I and VIII represent upper and lower bounds on system performance. Set VI is considered to be most realistic.

set	I/O block KB	arm access msec	bus speed MBS	disktime msec	IO_bust usec	L2_bust usec
I	4	8	1000	8.7	3.8	3.8
II	4	20	1000	21	3.8	3.8
III	4	8	512	8.7	7.6	7.6
IV	4	20	512	21	7.6	7.6
V	64	8	1000	18	61	3.8
VI	64	20	1000	30	61	3.8
VII	64	8	512	18	120	7.6
VIII	64	20	512	30	120	7.6

Model measurements of interest are as follows.

- Pi_{ut} : processor utilization

  the percentage of time jobs were being serviced by the processors, i.e. the time percentage the system did useful work. System performance is calculated as

  $$perf = Pi_{ut} \times 128 \text{ MIPS} \times nop$$

  where nop is the number of processors. Notice this is a batch environment definition of performance. Job throughput and mean response time, relevant in a database environment, is not a reliable result because of the initial assumption of Pitime = 80us = IFI_{up}. An accurate measure of IFI_{up} for a database transaction system is not known.

- $P0_{ut}$: supervisor utilization

- BUS_{ut} : bus utilization

In general bus and supervisor utilizations over 50% indicate potential bottlenecks (COCK81). Disk utilizations were not obtained because this measurement is highly inaccurate in a large simulation model.

2.3 Results

An extensive family of plots were generated, revealing samples of which are included as fig. 2.2 - 2.8.

Fig. 2.2 illustrates the results of parameter set VI. Each of the four plots shows degree of multiprogramming vs. performance at a constant miss ratio (vertical scales are different). The family of curves within each plot represent different numbers of simultaneously active disk arms. In other words, fig. 2.2 is a graph of the (nod,dmp,mr,performance) four space. Note the curves must all be monotonically increasing. Erroneously, the shared memory miss ratios modeled are not related to the I/O transfer size. The "true" miss ratio caused by a 64KB transfer size may be lower than the indicated one. Essentially the miss ratios must be interpreted with respect to the I/O transfer size although no analytic relationship is known.

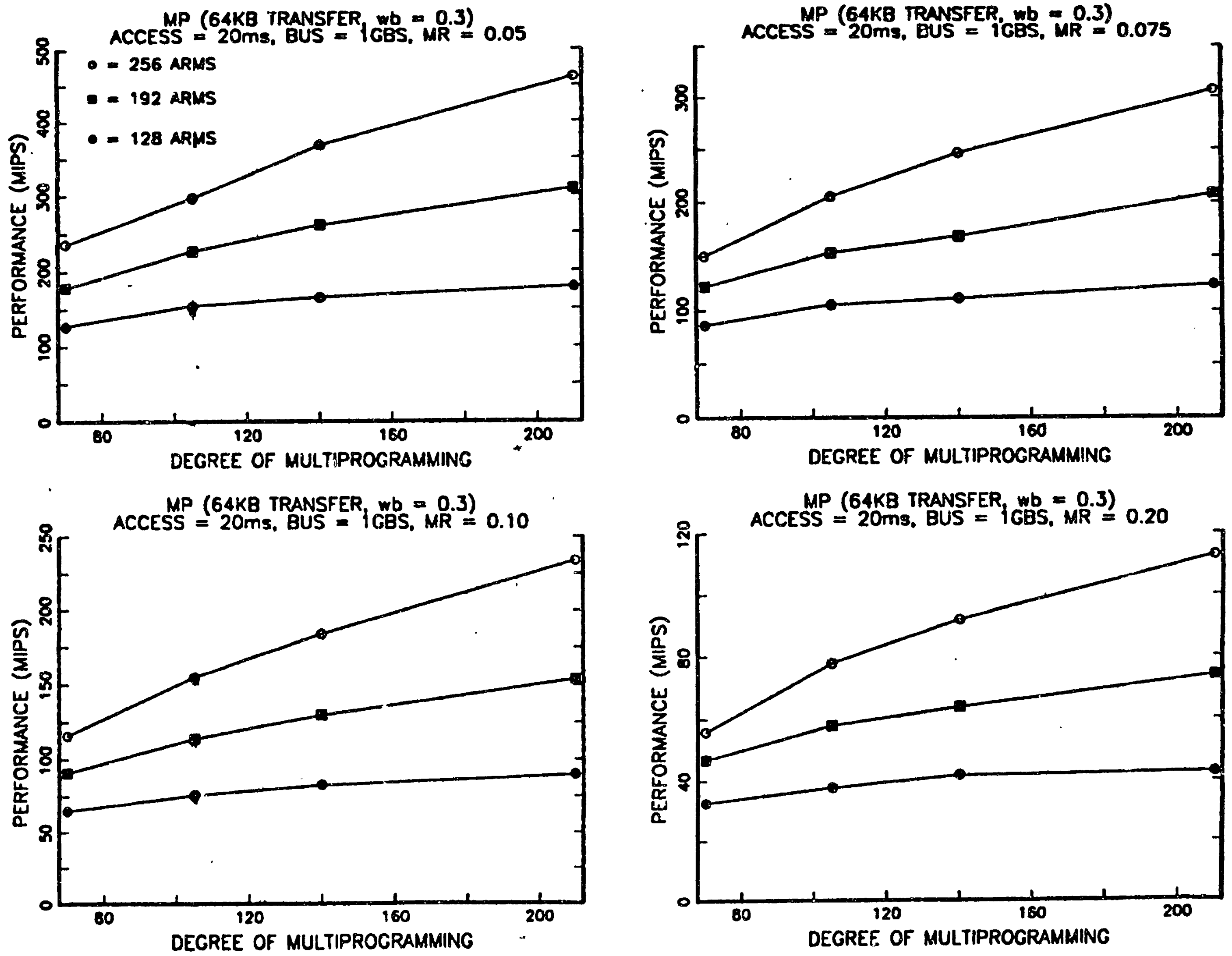


Figure 2.2 DISK ARMS vs. MISS RATIO

Figure 2.3. DISK ARMS vs. DMP

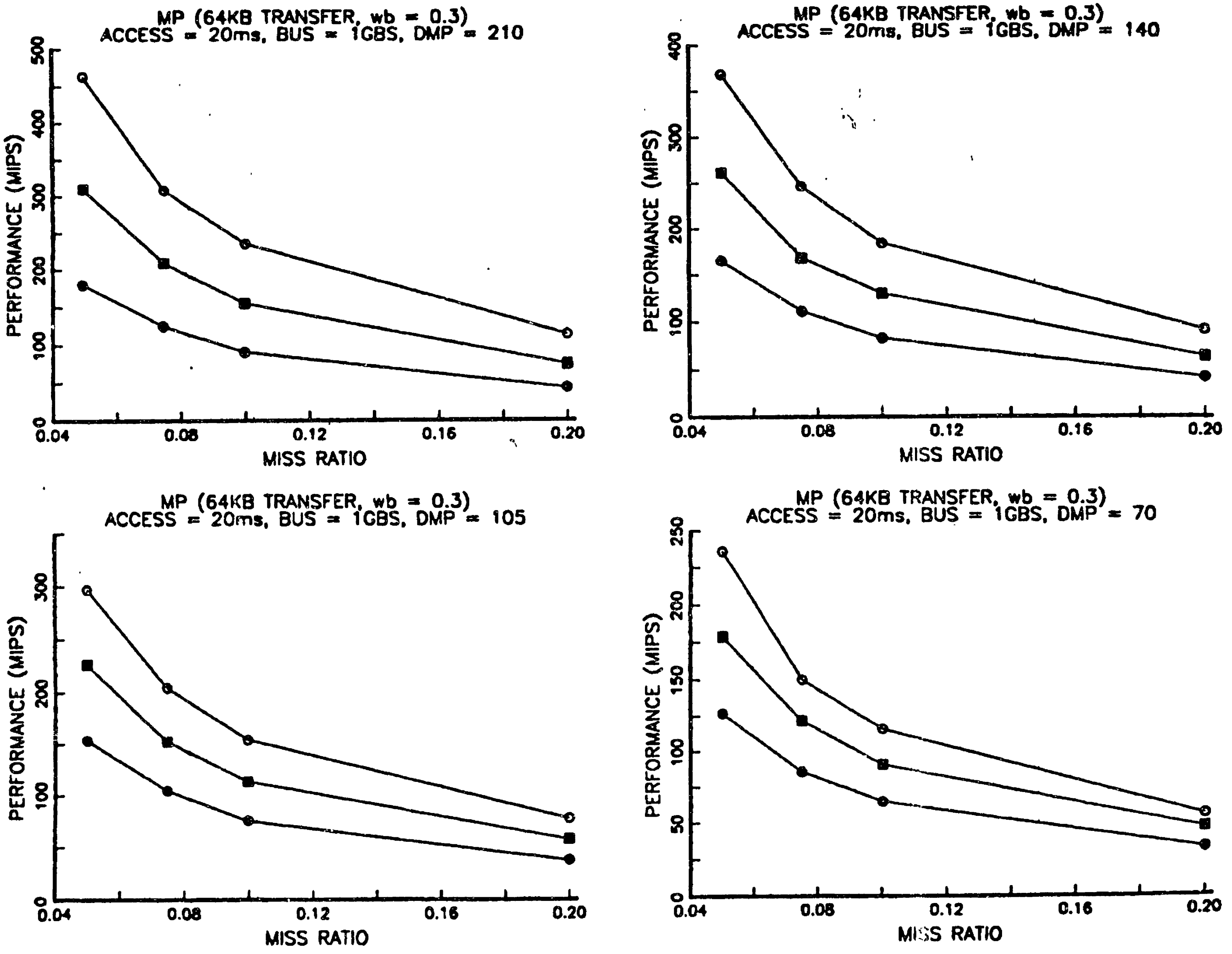


Figure 2.4. DISK ARMS vs. MISS RATIO

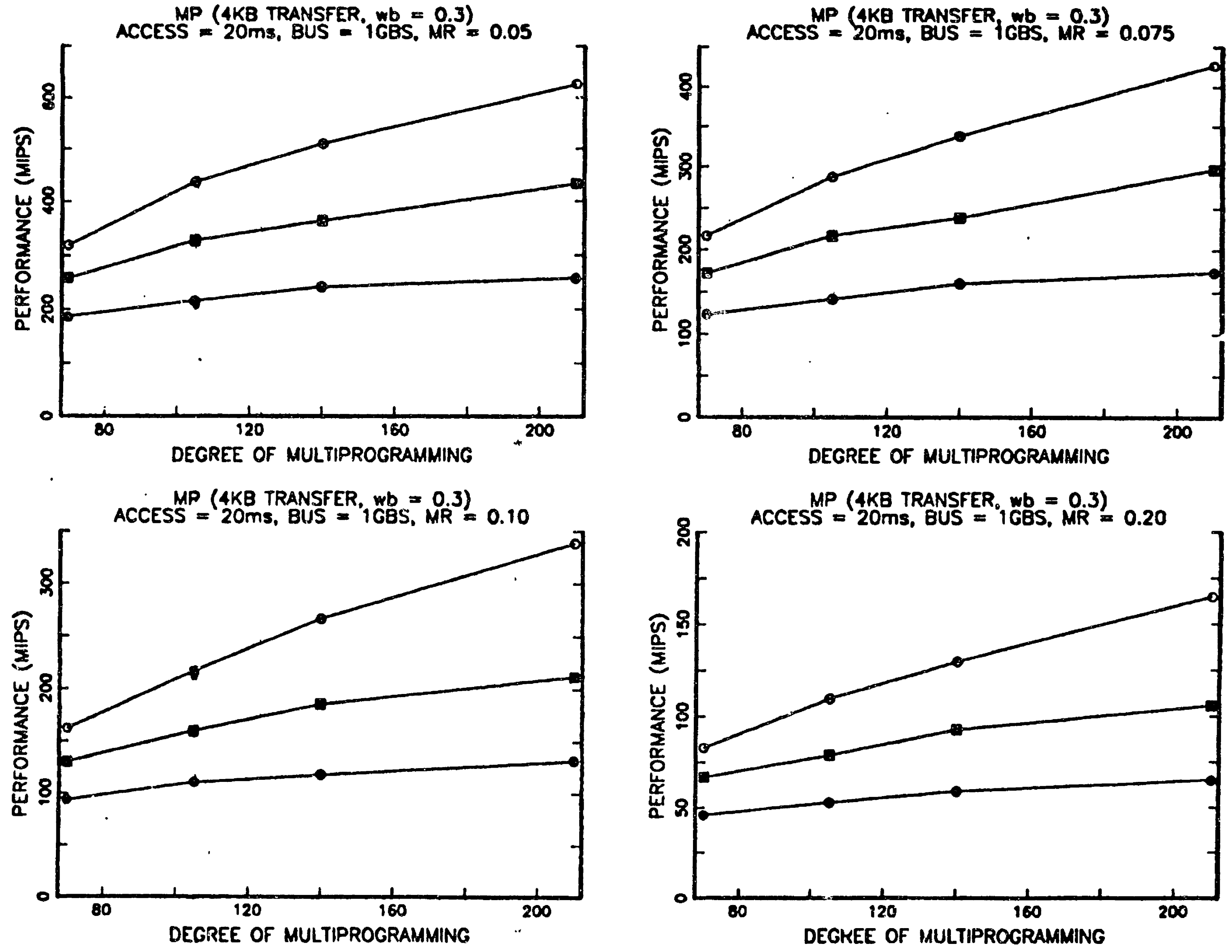


Figure 2.5. DISK ARMS vs. DMP

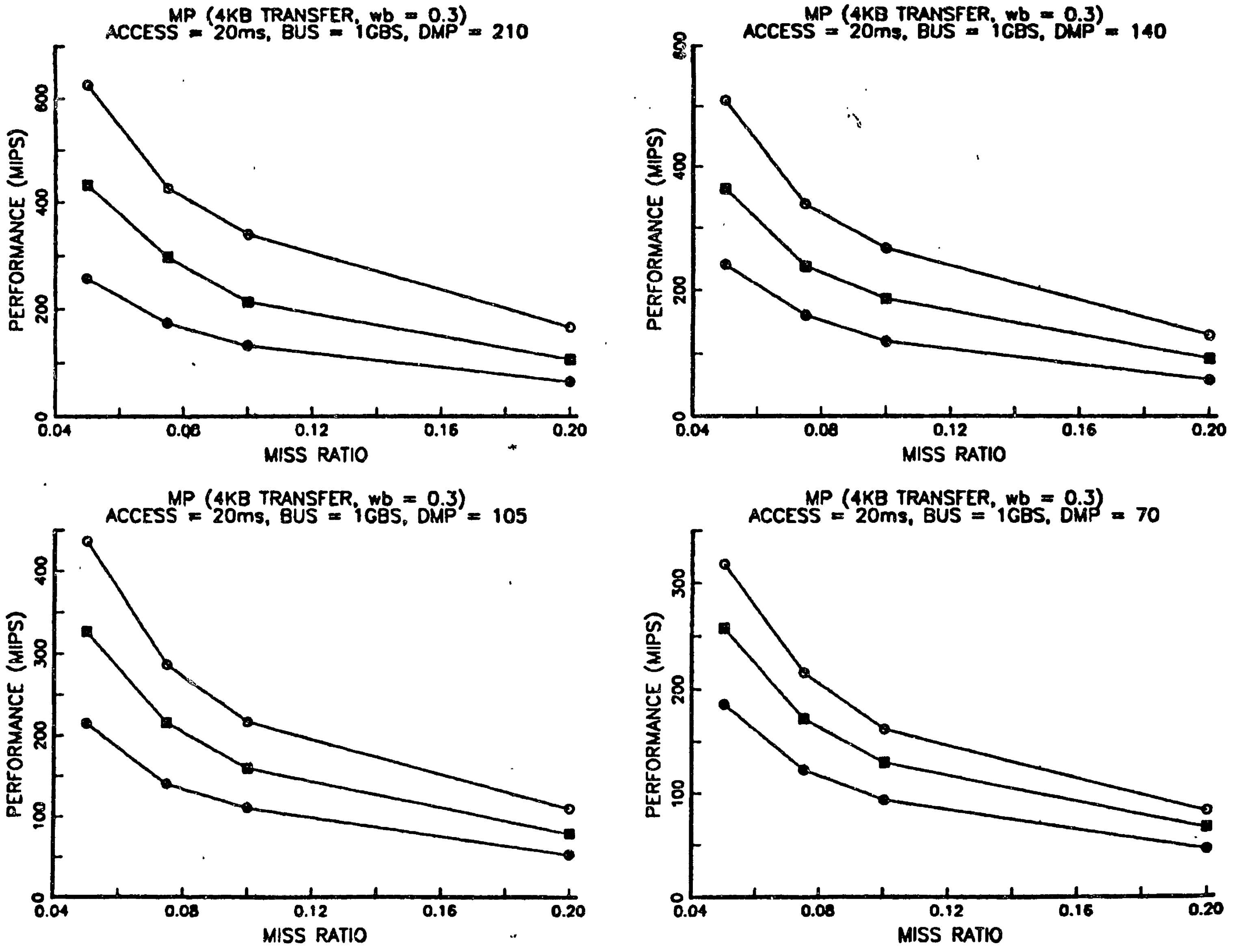


Figure 2.6. FAST DISK ACCESS (SET V)

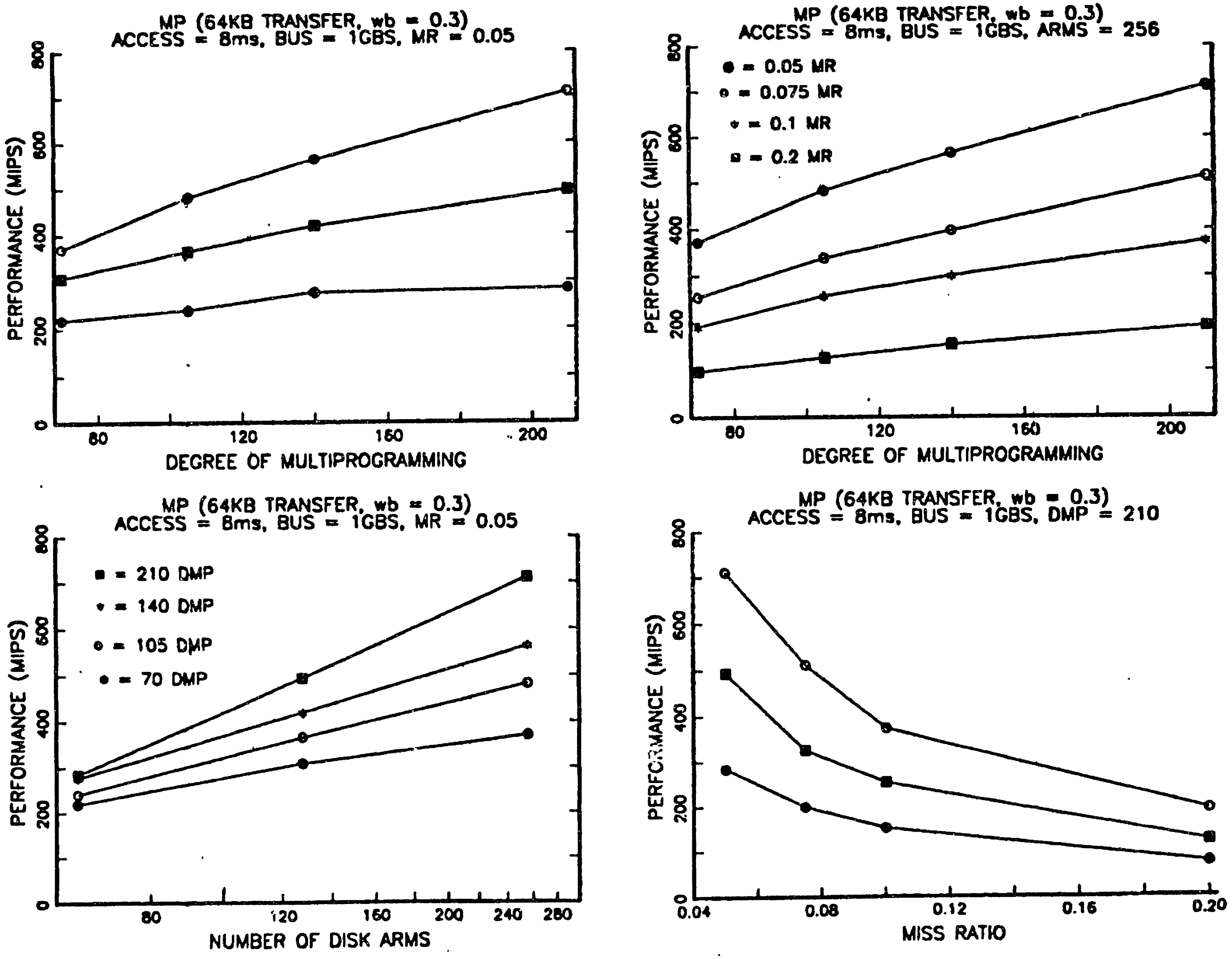
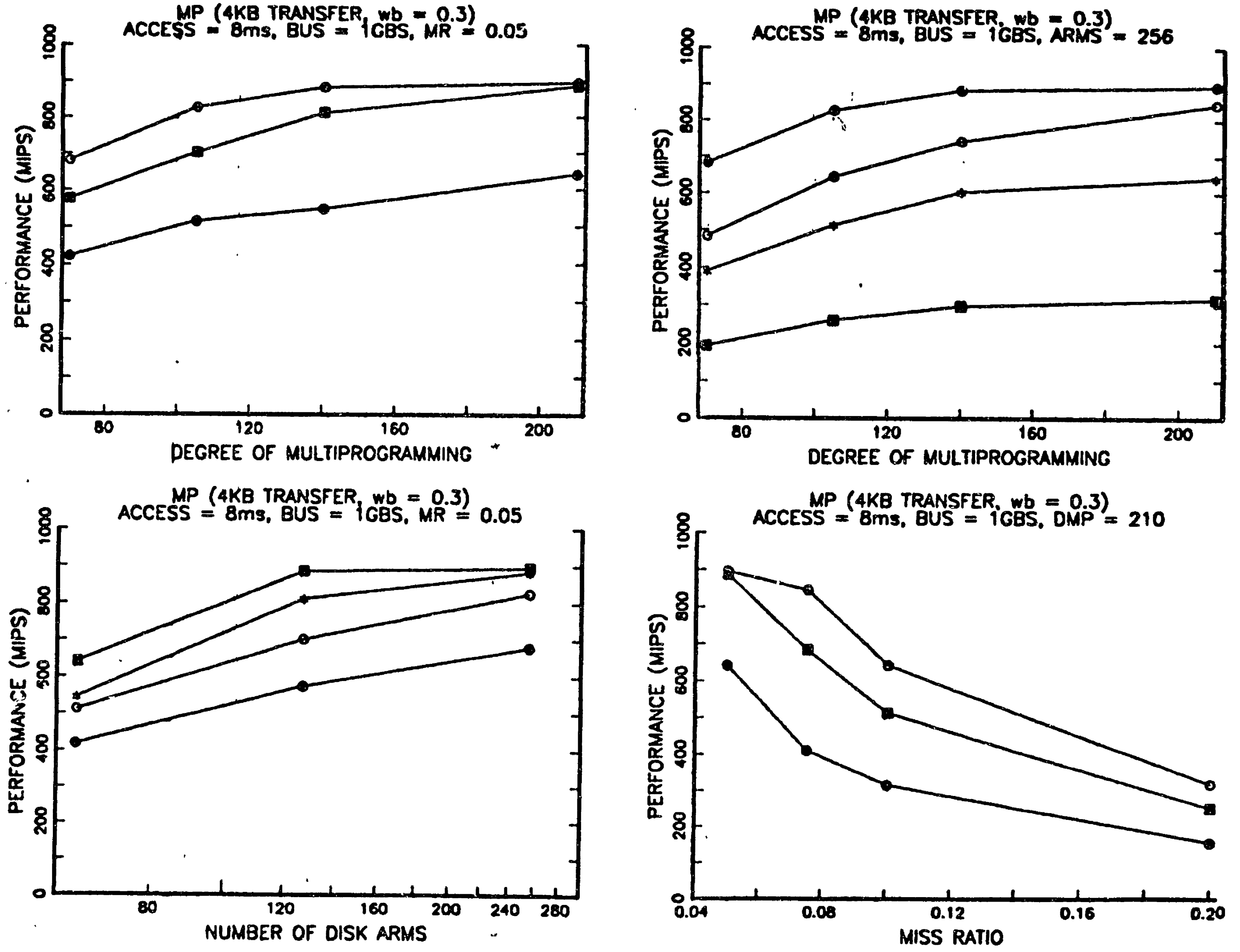


Figure 2.7. SET I – UPPER BOUND



The immediate observation from fig. 2.2 is that for any given miss ratio, many disk arms and a large degree of multiprogramming are needed for tolerable performance. The dmp cannot be easily increased. Increasing the size of Ms will permit a large active transaction set, but how large is unclear. For a given performance, the required disk arms and dmp can be reduced if the shared memory miss ratio is reduced. This is most clearly shown in fig. 2.3 for families of miss ratios. The miss ratio cannot be easily decreased either. A third solution would be to increase the number of disk arms. This is unappealing because it is too expensive.

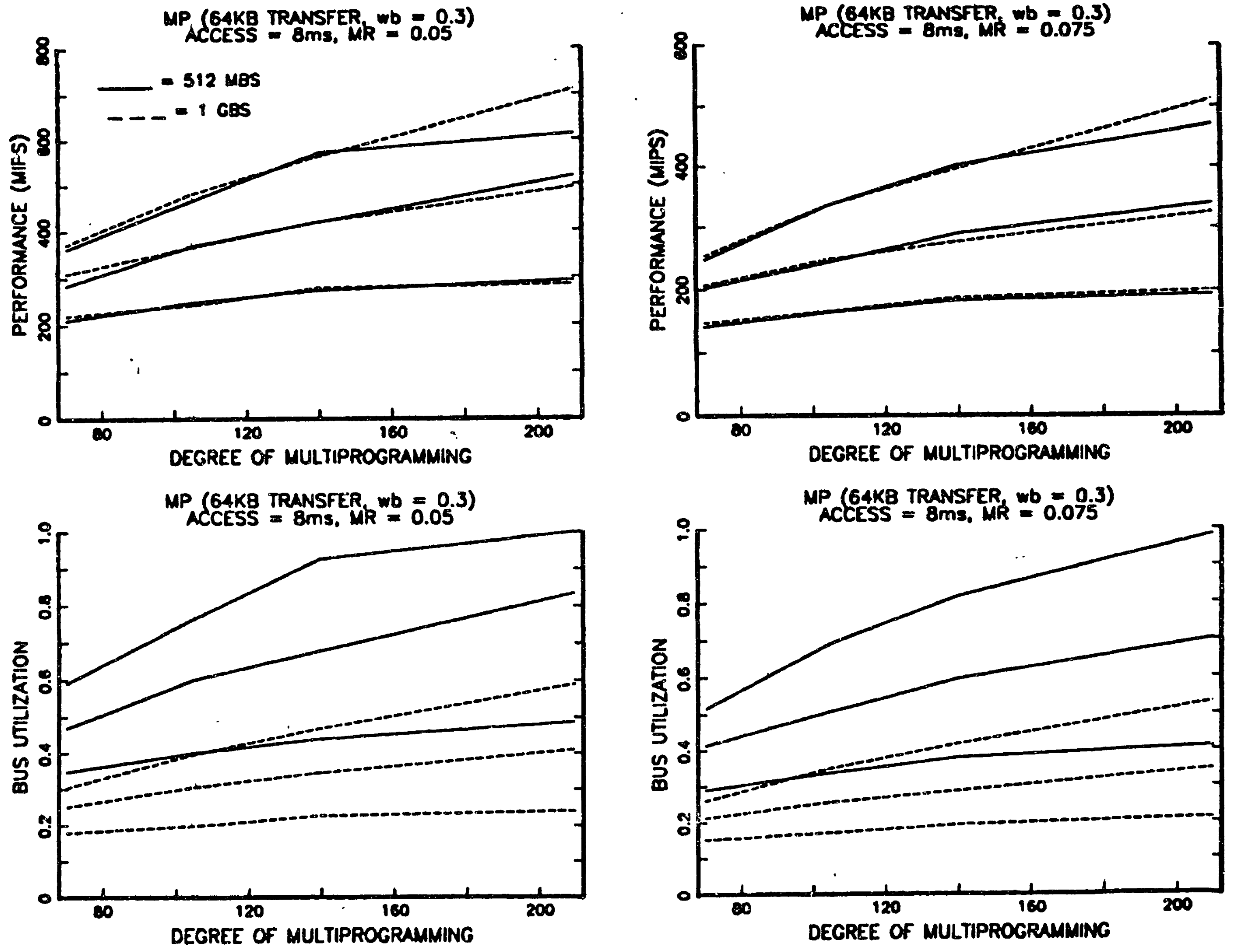
Fig. 2.4 and 2.5 illustrate similar results of parameter set II. The smaller 4KB transfer alleviates the problems to a certain degree, but does not solve them. The miss ratios must be interpreted with respect to the 4KB transfer.

An alternative is to push disk technology is produce a lower disk arm access time (BRAN81b). The results for parameter set V are shown in fig. 2.6. The disk arm and dmp constraints are lessened to acceptable levels. Note, however, that the miss ratio must remain low. This suggests that the shared memory level is essential because *a high fault rate between the processors and secondary memory will degrade performance drastically no matter what.*

Fig. 2.7 illustrates the results for parameter set I - the upper bound on system performance. The small I/O transfer size is reflected in the high processor utilization. Note that as few as 140 active transactions and 130 simultaneously active disk arms can produce near maximum performance when mr = 0.05. However the actual shared memory miss ratio is relative to the I/O transfer size. Thus the modeled miss ratio of 0.05 given a 4KB transfer may be a significantly higher true miss ratio, corresponding to lower actual performance.

Fig. 2.8. compares resource utilizations resulting from parameter set V and VII, i.e. a 512MBS bus (solid lines) and the 1GBS bus. System performance is approximately equal for both except at high degrees of multiprogramming. Bus utilizations are vastly different. The high bus usage of the slow bus cannot be tolerated. Notice that the fast bus remains below 60% utilization for all parameter values.

Figure 2.8. SLOW BUS (SET VII)



3. System Architecture

The hierarchical addresses are summarized in figure 3.1. The assumed virtual space is 2^{44} B = 16384 GB. This space is partitioned into a maximum of 32K ***segments***, each a maximum size of 8K pages, where virtual page size is 64KB. This allows a maximum segment of 512MB. The virtual address space is the only one in the system, i.e. it is shared by all processes, in contrast to conventional multiprogrammed computers where each process has its own virtual space. The processes here divide the virtual space among themselves by either *sharing* or *exclusively owning* the segments. This sharing and exclusion refers to *file protection* and is not to be confused with a lower level sharing/exclusion dealing with *consistency*. Even protect-shared segments

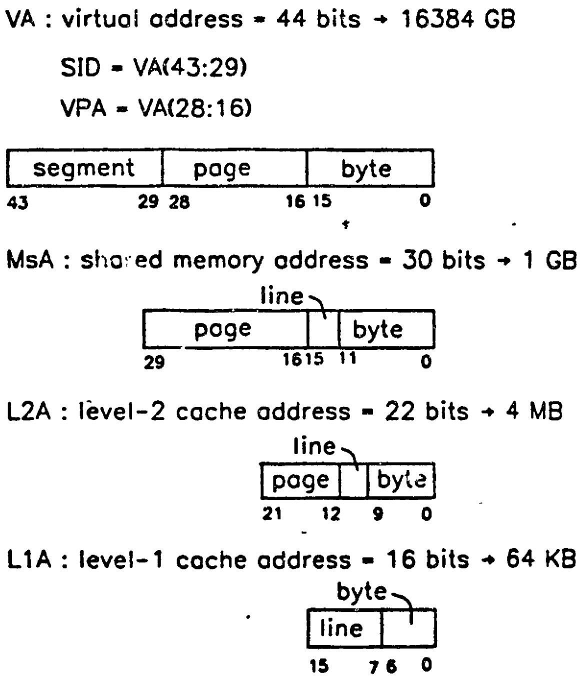

VA : virtual address = 44 bits → 16384 GB
SID = VA(43:29)
VPA = VA(28:16)
segment
page
byte
43
29 28
16 15
0
MsA : shared memory address = 30 bits → 1 GB
line
page
byte
29
1615
11
0
L2A : level-2 cache address = 22 bits → 4 MB
line
page
byte
21
12
9
0
L1A : level-1 cache address = 16 bits → 64 KB
byte
line
15
7 6
0

Figure 3.1. Hierarchical Addresses

can be exclusively "locked" during an inconsistent phase of a transaction (section 8.1). Such a "lock" is *not*, however, as permanent as a protection "key".

The term ***file space***, as used in this thesis, refers to the in-use virtual space. The proposed architecture equates a file with a segment. Therefore a user file, defined by a unique pathname, can be referred to by one and only one ***virtual segment identifier***, SID. The issues of machine instruction address to virtual address translation are not dealt with in this thesis.

The following are necessary architected features of the BIP MP system. This is not a complete set but rather the few important concepts that relate to the memory hierarchy organization and control. The rest of this section should be skipped over until referenced in the text.

- Forcing a data object from the private processor memory to the shared memory is architected with the following machine instructions

  ```
  INTP   PID,VPA
  INTL   PID,VLA
  ```

  where PID is a processor identifier, VPA is a virtual page address and VLA is the virtual address of a 4KB data object. **Interrogate Page,** INTP, causes the interrogation of processor PID for any copies of the page addressed by VPA. **Interrogate Line,** INTL, is used to interrogate for a virtual 4KB object, i.e. an Ms line possibly occupying L2 page frame(s). Both instructions are implemented by the same hardware in the processor private memory. The only difference is that INTP will take 16 times longer than INTL, worst case. If the CPU is currently manipulating part of the object specified, the interrogate waits for CPU completion and then prevents further accesses of those data objects. The data objects in L1 are written back to L2 if they have been modified. Those L1 lines are then invalidated (L1DIR is updated). The data objects in L2 are written back to Ms if they have been modified. Those L2 objects are then invalidated (L2DLAT and L2DIR are updated). The interrogate instructions are necessary to implement the COMMIT, ABORT, recovery and Ms page fault handler procedures.

- Manipulating entries in the L2 map is accomplished by referencing L2HIT and L2DIR by their virtual addresses. These segments are not accessible in problem state. This eliminates the need for locking them if the concurrent operating system procedures which modify the maps do not attempt inconsistent actions. There is no problem with L2 references because they are only made while the associated processor is halted during local recovery (section 9.2).

- The Ms map store is not logically part of the shared memory, although MsHIT, MsDIR and the shared memory access counts belong to a single segment in the virtual space. The existence of the Ms map store is not transparent to the supervisor, which manipulates the map with the following machine instructions

```
MLL   GPR,LA
MU    LA
MSU   GPR,LA
```

where LA is a 16B address relative to the map segment and GPR is a general purpose register in the CPU. **Map Lock and Load**, MLL, locks a map entry by setting its hardware lock bit and then loads the entry directly into a CPU register. If the entry is already locked, the instruction waits for it to be unlocked, then locks and loads it. **Map Unlock**, MU, is used to unlock an Ms map entry. **Map Store and Unlock**, MSU, is used to modify an entry and then unlock it. These instructions are used to update the Ms map in a consistent manner. The lock bits prevent the hardware mapping unit, MsC, from making concurrent updates to the same map entry (section 4.4.2). Access count entries (64 two bit counts per Ms map store entry) are also manipulated by the software with these instructions.

- Machine instructions must be architected and supported by hardware facilities for cold starting the system. This involves initializing sections of the memory maps and loading areas of shared memory.

- Machine instructions must be architected to initiate I/O to secondary memory. Channel control programs are given the MsA of the I/O buffer frame in shared memory and the physical address of the target data object.

4. Memory Management

The proposed memory hierarchy is composed of four major levels: level one cache - L1, level two cache - L2, shared memory - Ms, and secondary memory - Bs (fig. 4.1). Bs is referred to as the top of the hierarchy. L1 and L2 comprise the processor private memory. Both the L2-Ms and Bs-Ms interfaces are connected by the same system bus because Ms is a single port memory (section 7.1). Issues concerning tertiary memory and access-gap filler (REGE76, POHM75, LANG76, BRAD80) will not be discussed.

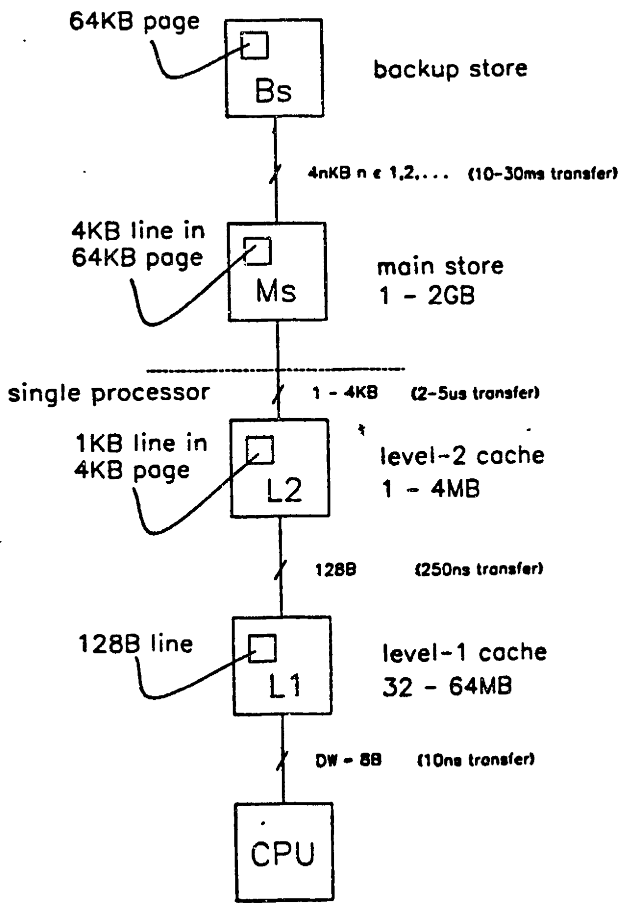


Figure 4.1. Memory Hierarchy Overview

It is assumed a CPU makes two types of memory ***accesses*** or ***references*** - loads and stores. References are always for a single word although a double word containing the target word is delivered to the CPU. The hierarchy is ***store-in*** as opposed to ***store-through.*** Thus when L1 misses on a store reference, the data word is not permitted to bypass L1 and be stored directly in L2. Instead, the target must be transferred to L1 where the store occurs. This policy implies that each level L_i in the hierarchy will contain a subset of the data objects contained in L_{i+1}, and that *the most up-to-date copies of data objects are at the lowest levels.* Data objects, purposely left as an abstraction because of this, are not passed down the hierarchy in totality. Each level L_i is comprised of ***frames*** (page frames), large enough to hold an L_i ***page.*** An L_i page is composed of L_i ***lines.*** Within the lowest and highest levels, L1 and Bs, the page and line sizes are equal.

For the intermediate levels, data objects are allocated frames, but transferred in as a multiple of the line size. For example, a 4MB L2 contains 1024 page frames, each 4KB in size. Each frame holds four lines, each 1KB in size. A request for a 4KB data object not presently in L2 will eventually cause that target object to be allocated an L2 frame, and either 1, 2, 3 or 4KB will be transferred. If the next line is accessed, it is loaded from Ms into the same frame if not already present. Frames therefore have valid and invalid or undefined lines in them. The reasons for this policy are to both reduce the mapping overhead by allocation on large thus fewer frames, and yet to retain low miss *time penalties* by transferring short lines.

Reducing the map size is the predominant design factor, especially since L2 must be fully associative and Ms is so large. It is not clear how badly the miss ratio of level L_{i-1} will be affected by the L_i policy of line instead of page transfers. This uncertainty is alleviated in the Bs-Ms transfer by explicit I/O and intelligent demand paging and pre-paging mechanisms which issue several line transfers at once. The Ms to L2 transfer is adjustable from 1 - 4KB. The actual implementation retains information about 1KB lines [1].

Each level L_i in a memory hierarchy is characterized as follows (YEN81).

- mapping algorithm
- replacement algorithm
- update policy
- fetch policy
- homogeneity
- size
- page size
- line size
- transfer (bus bandwidth)

As mentioned, the update policy is store-in at each level. Although a store-through policy avoids the replacement problem, it causes higher traffic between levels. (SMIT79) claimed that store-in is the better policy if additional staging buffers are placed between levels. This scheme is described below. Fetch policy is pure demand paging, i.e. fetch any data object that misses and only when it misses. Ms also has a pre-paging policy for sequential files (section 7.3). Data objects are inhomogenous, i.e. they can contain both programs and data, at each level. Page and line sizes are discussed above.

A level L_i mapping does two things. The mapping first determines if a "target" data object is resident in L_i. If it is, L_i ***hits*** if it is not, L_i ***misses.*** The second map function is to determine ***where*** the target is (which line and frame in L_i, i.e. an L_i address) if the mapping was a hit. In some instances, the map is itself organized as a

[1] The L2 design is theoretically a ***sector buffer organization*** (RAO78), with many ***sectors*** (L2 pages) and a small number of ***blocks*** per sector (L2 lines per page).

hierarchy. Consider the case of a two level map. The lower map level is implemented in a faster technology than the upper map to decrease access time and is smaller to reduce cost. The lower map can hit or miss depending on whether it holds the information necessary to determine the L_i hit/miss. If it misses, the upper level map must be accessed. The update policy of a map hierarchy is usually store-through, i.e. the complete map must always be consistent. The mapping must be resolved before a hit/miss in the corresponding memory level can be determined.

A mapping is therefore a translation or transformation of a source address to a target address. The source address space is the mapping's ***domain*** and the target space is the mapping's ***codomain***. The mappings in this system, viewed in this mathematical sense, are not injective, simply a result of the domain being larger than the codomain.

L_i replacement refers to finding a suitable L_i page frame to evict from L_i if or when an empty frame is needed. Replacement is invoked to allocate room for target objects which missed in L_i. The replacement policy can play a major role in the control of activated processes, or it can be passive. In general, lower levels in the hierarchy cannot afford sophisticated replacement algorithms and therefore do not use working set criteria. Levels higher up, such as Ms, can have complex policies, possibly implemented in software, because timing is less critical; however because the number of pages is larger, the cost of implementation may increase.

L_i transfer refers to the physical transfers requested by L_{i-1} after an L_i hit, by L_i itself after an L_i miss and by L_{i+1} during an interrogate. Conceptually, each level makes upward mapping and transfer requests and performs both upward and downward transfers (fig. 4.2). Upward transfers are called ***write-backs*** and are necessary only when the data object being replaced or interrogated is "dirty", i.e. has been modified. Both levels involved in a transfer have their memories tied up. In addition,

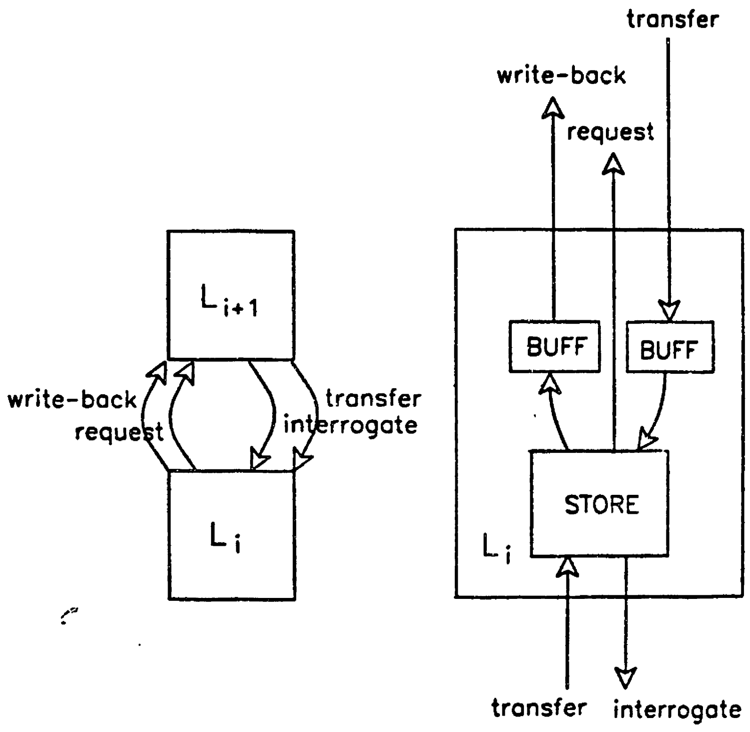


Figure 4.2. General Memory Level

a downward transfer may require a previous write-back to clear a frame for the downward transfer's data object. To avoid this double penalty, buffers are placed between the memory levels, permitting both upward and downward transfers to proceed in parallel [1]. In theory these constitute additional memory levels with simple characteristic~. In this thesis, buffers are grouped with the memory level below them.

Rough service times are given in fig. 4.1 for a hit in each level of the memory hierarchy assuming no contention. L1 can deliver one DW in the L1 access time of 10ns. L2 can deliver a 16 DW L1 line in

$$\begin{aligned} service &= latency + mapping \\ &= 75ns + 170ns = 250\ ns \end{aligned}$$

[1] The inclusion of buffers makes the transfer policy a derivative of **Flagged Register Swap**, FRS, described in (POHM75).

assuming a conservative packaged L2 store access time of 75ns (50% increase over chip access). Ms can deliver one 1KB L2 line in 2us and four L2 lines in 5us (section 1.2.1). Bs can deliver a 4KB Ms line in

$$\begin{aligned} \textit{service} &= \textit{latency} + \textit{transfer} \\ &= 20\textit{ms} + 700\textit{us} = 21\ \textit{ms} \end{aligned}$$

assuming I/O initiation (section 6), replacement decision (section 4.5.1.2) and mapping times (section 5.3) are insignificant.

The relative sizes of the hierarchical levels are difficult design parameters to pinpoint without a detailed timer. L1 is 32 - 64KB, the size of a conventional high performance UP cache. L2 is 1 - 4MB, smaller than a conventional primary memory (32 - 64MB) for two reasons. First, L2 maps must be small enough to facilitate hardware control. Second, Ms is designed rather large to compensate for L2 smallness. Ms is 1 - 2GB, designed to hold roughly 1% of the in-use file space which is approximated as 1% of the virtual space (COCK81). Comparisons between the mappings used in the proposed system and in conventional systems are given in fig. 4.3. An overview of the memory mappings is given in fig. 4.4.

4.1 L1 Mapping

L1 is mapped much like an IBM 370/3033 cache, except that there is no non-relocate or REAL mode wherein a shared memory address, MsA, is mapped to an L1 address, L1A. Only *relocate* or *direct address translation,* DAT, mode, is supported wherein a virtual address, VA, is mapped to L1A. Non-relocate addressing, useful in diagnostics and cold start-up, is performed by special hardware which *bypasses* the caches. It is argued here that an efficient mapping is not needed for non-relocate

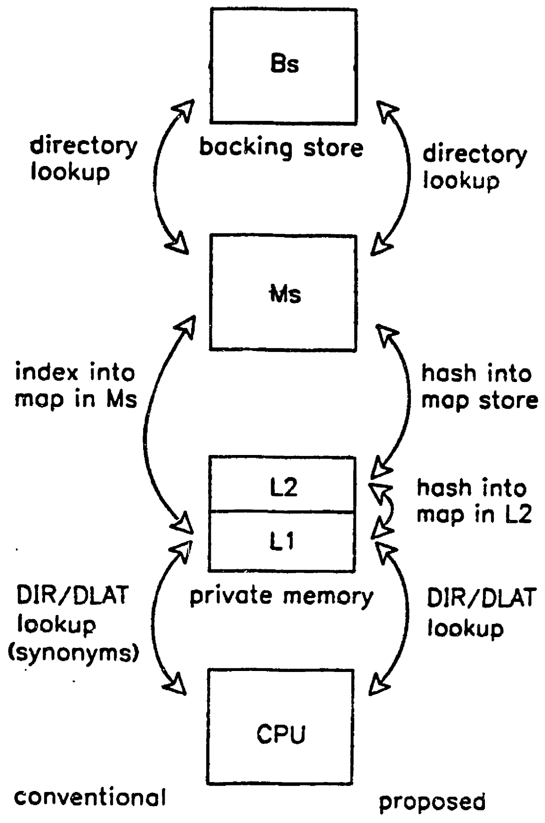


Figure 4.3. Mapping Comparisons

because this mode of addressing should not be necessary in any systems or applications program:

- It can cause synonyms.
- Operating systems usually need to find the real frame address of a virtual address for I/O purposes, for instance when using a real addressed channel device. I/O buffers can alternatively be mapped like all other segments, if the buffer can be "frozen" in Ms. This facility is provided in the proposed system.
- User programs should never use real addressing. It is bad programming practice.

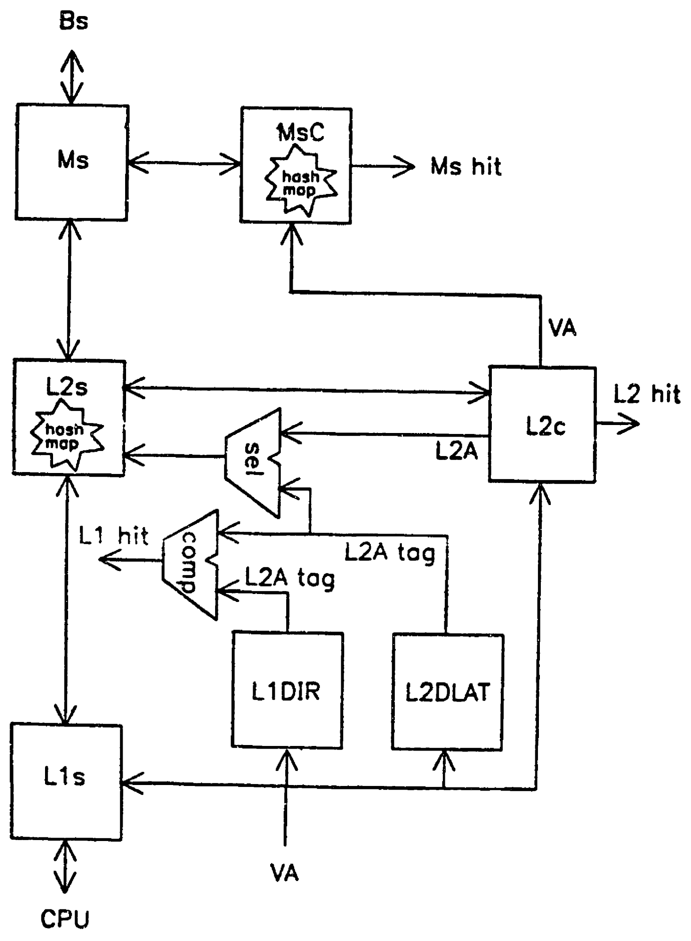


Figure 4.4. L1, L2 & Ms Mapping Overview

- System bootstrapping usually needs to load real frame addresses. An alternative is to provide a hardware driven bootstrapping method.

Two designs are presented for L1: 4KB and 64KB virtual page size. The impact of virtual page size on cache design lies in the possibility of aliasing or synonyms. This problem cannot arise in an architecture with no REAL addressing mode however. The virtual page size is equal to the Ms page size so that the virtual pages loaded from disk fit into the memory frames at the highest level of the hierarchy. Thus virtual page size influences mapping and replacement algorithms for Ms. It is assumed throughout this discussion that the L2 page size is fixed at 4KB. The proposed system has a 64KB virtual page size for reasons concerning Ms mapping algorithms (section 4.4).

4.1.1 Conventional Map

A conventional memory hierarchy has two levels of volatile memory: a level-1 cache, L1, and a main store, Ms. The following is a description of one standard mapping scheme for this simple hierarchy.

Assume L1 is 4-way set associative - the cache and tags are divided into 4 sets (columns), 64 lines/set. Each row is a congruence class, i.e. the mapping will route mutually exclusive subsets of the real address space to these classes. Within a class, however, any of the 4 sets may hold the desired or *target* data object. The ***L1 directory***, DIR, holds a mapping entry for each line of each set. This entry contains information to determine if the line searched for is cached. The map is operated in two different modes: REAL and DAT. The mapping for REAL mode translates an MsA to L1A. The DIR is addressed with MsA(12:7) [1]. MsA(m:7) is the real 128B line address and MsA(6:0) is the byte address within a line. The addressing is done with 6 bits, giving one of 64 congruence classes.

Each class holds 4 set entries, so additional information is needed to select among them. The additional information used is the remaining real address bits (sometimes called the tag or key), MsA(m:13), stored in each entry and compared against the target real address in parallel to the cache access. If one comparison matches, the cache hits, i.e. the accessed data is correct. Two or more comparisons cannot match because cache update policy is to never load multiple copies of the same data object.

[1] HEX notation for a binary field, e.g. a 1GB Ms is addressed with MsA(m:0) where m = 29 and is the most significant bit of the address (DENN81).

DAT mode, the mapping from VA to L1A, is implemented in a similar manner. However there is no longer an MsA to compare against the DIR tags. That is where the ***directory lookaside table,*** the DLAT, comes in.

The DLAT is a 2-way set associative map between virtual and Ms addresses. It is used only in DAT mode addressing. The DLAT is addressed by the low order 7 bits of the virtual page address. That is VA(18:12) for a 4KB page and VA(22:16) for a 64KB page. The entry requires further information to choose between two sets within a class (there are 128 congruence classes). Thus VA(n:19) is kept for 4KB page (VA(n:23) is kept for 64KB page), where n is the most significant bit of the address, and compared against the target virtual address. This is the complete high order address, including segment number. If the DLAT entry key matches, then the MsA in the entry is valid. If the key doesn't match, the DLAT misses. Thus in DAT mode, *three accesses are performed in parallel:* the cache, DIR and DLAT. To determine if one of the four sets of the cache access is valid, the DLAT must hit and one of the four DIR sets and the DLAT MsA fields must match. Unfortunately, things get trickier than this.

A virtual page size of 4KB sets the number of identical address bits in VA and MsA at 12, i.e. VA(11:0) = MsA(11:0). This is because Ms and virtual page sizes are equal. In other words, data objects *within* a page are directly addressable, whereas data objects larger than a page are possibly mapped to several disjoint Ms frames. Recall that the DIR is addressed with the target MsA in REAL mode. In DAT mode the DIR must be addressed with the VA because the MsA is not known at the time of access. If the DLAT was mapped *before* the DIR, then this problem would be avoided; however both accesses are performed concurrently for speed. Unfortunately $VA(12) \neq MsA(12)$. Thus if it cannot be guaranteed that the line was placed into the

DIR using a unique address (virtual or real), both ***synonyms,*** VA(12:7) and ¬VA(12) | | VA(11:7) [1] must be used to address the DIR and each resultant class must be compared, in parallel, to two DLAT classes, accessed with the same synonyms. If either one matches then the cache hits.

Another way to view this ***aliasing*** problem is to assume that the data object was originally mapped into the cache by its real address, i.e. it was first accessed in REAL mode. The object may now be accessed in DAT mode, the user supplying its VA only. This address contains only enough information to pin down the object's location in cache to within two classes (corresponding to one synonym bit). Each location must be checked in the normal manner, i.e. against a DLAT produced tag. The tag size must be increased to MsA(m:12) because both synonyms may have equal MsA(m:13) tags. Note that the number of synonym classes which must be checked is equal to 2^N where N is the number of synonym bits. Therefore whereas one or two synonym bits can be tolerated, e.g. IBM 370 Series, more cause intolerable mapping inefficiencies.

Another instance of aliasing is if the architecture permits a data object to be named with more than one virtual address, e.g. System 370. In this case, the object could be mapped into the cache using one of its synonyms and then accessed using the other, both in DAT mode. This type of architected aliasing is not dependent on the mapping organization and is avoided in the BIP MP (section 3).

If the map never operates in REAL mode (and there is a one to one relation between data objects and virtual addresses) aliasing cannot arise because data objects are always placed into the cache in DAT mode, a mapping done on VA(12:7), a unique identifier for each object. The tag must still be MsA(m:12) because the full

[1] HEX concatenation operator.

Ms 4KB page address is needed when L1 misses. For a 64KB virtual page size, there is no aliasing either, but for a different reason: VA(15:0) = MsA(15:0). In this case, tag size reduces to MsA(m:16).

In general there are two necessary conditions for the creation of synonyms stemming from map organization and architecture. The map organization problem arises between any two memory levels, L_i and L_{i+1} from two factors: L_i set size and L_{i+1} page size. If the L_i set is larger than the L_{i+1} page, the potential for synonyms exists from the overlapping bits (fig. 4.5). The map architecture problem arises if the user references a data object by more than one unique address.

The MsA was originally needed in an L1 mapping to perform REAL address translation to L1A. The DLAT was then introduced to hold the MsA for determining a DIR hit during DAT mode. Now assuming REAL addressing is not required, any common identifier can be used for the DLAT/DIR tag; however, the MsA is still needed in the DLAT to access main memory on an L1 miss. This will be an important consideration when dealing with a multi-level hierarchy.

To summarize, if the DLAT hits then there is a good chance the data object is in cache, however that is not guaranteed. If the DLAT misses then there is small chance that the data object is in the cache. Therefore when the DLAT misses, it is updated (from the Ms map, to be discussed later) and the correct MsA is then compared against the DIR tags. If the DIR now misses, then the cache does not hold the target object.

Different bits in the virtual address are mapped to the DLAT and DIR, thus these mappings do not correspond in any intuitive sense, i.e. the map update protocol is not dependent on the mapping transform. In addition, the DLAT and DIR have different

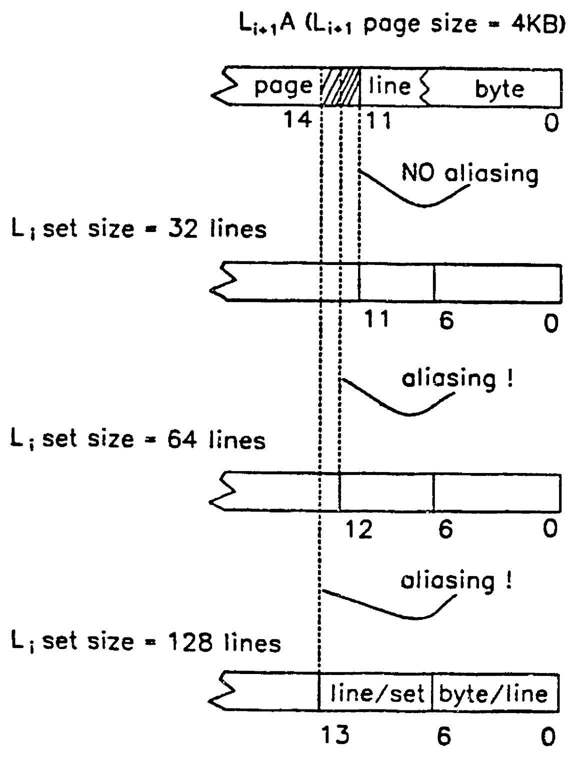

$L_{i+1}A$ (L_{i+1} page size = 4KB)
page
line
byte
14
11
0
NO aliasing
L_i set size = 32 lines
11
6
0
aliasing !
L_i set size = 64 lines
12
6
0
aliasing !
L_i set size = 128 lines
line/set
byte/line
13
6
0

Figure 4.5. The Synonym Problem

set associativities and thus different replacement policies. The fact that both tables have 64 × 4 = 128 × 2 = 256 entries is an implementation decision *only*. The DLAT is 2-way instead of 4-way to save tag comparators, with no substantial loss of performance. Theoretically, the DLAT can be any sizc, independent of the size of the level-1 cache. Because of this there are no necessary constraints between the DLAT hitting and the cache holding the target object; however, with the DLAT large enough, a sensible replacement policy and no degenerate reference patterns (such as accessing the Nth page of consequtive segments) a DLAT miss almost always implies a cache miss [1].

[1] A cache hit can be *defined* as a DLAT and DIR hit, in which case a DLAT miss implies a cache miss. This is not the definition used in this thesis.

4.1.2 Hierarchical Map

The major difference between the proposed system's memory hierarchy and a conventional one is the addition of a second level cache, L2. This cache is the highest level in the private memory of a processor, i.e. it plays the role of a conventional main store. Thus L2 is the target level of an L1 miss, implying that the DLAT codomain is the L2 address space and the DLAT tag must be L2A(m:12). The DIR tag will be compared against the corresponding DLAT tag and therefore must be L2A(m:12) also. The names of these maps are clarified as the L1DIR and L2DLAT, corresponding to their targets. Real addressing, i.e. user addressing with an MsA, is not implemented in the BIP MP. REAL mode would require an *additional* DLAT addressed by MsA, L2DLAT*. L2DLAT cannot be used because it is addressed by $VA(22:16) \neq MsA(22:16)$, i.e. the entries are not placed properly for real addressing. L2DLAT returns the L2A of a virtual data object. L2DLAT* returns the L2A of a "real" data object, i.e. a data object resident in Ms. For real addressing L2DLAT* would be mapped instead of L2DLAT and used if L1 misses to give the L2A. If L2 misses, the user MsA is used to access the shared memory. Both L2DLAT and L2DLAT* would have to be updated concurrently on L2 replacement. The cost of L2DLAT*, in addition to the undesirable design constraints needed to avoid synonyms, are valid reasons to avoid real addressing.

This thesis does not deal with L1 design and therefore nothing more detailed than simplified L1 interfaces are given in Appendix B. Note that L2DLAT is considered part of L1 for historical reasons. L1 design and implementation issues for set-associative and other organizations are well documented (section 1.3).

L1DIR and L2DLAT entry definitions are given below.

L1DIR entry

	bits	
TAG	10	tag : L2A(21:12)
DL	1	dirty line flag
VE	1	valid entry flag

L2DLAT entry

VK	25	virtual key: VA(43:19), the portion of VA not used in indexing L2DLAT
TAG	10	tag : L2A(21:12)
PK	2	protection key
VE	1	valid entry flag
VL	4	valid line

The DL flag is set on a store into the associated L1 line. This dirty bit indicates that the line must be written back on replacement. More subtly, it indicates that the L1 line is the most up-to-date version of that 128B in the entire system. PK holds the file protection key for the data object. Read-only, R/O, read-write, R/W, and execute-only, E/O, are considered in this thesis. On the L1 level this field is used to determine if a reference is legal. The VL bit vector is not necessary for the L1 mapping, but decreases the L1 miss penalty by indicating if the data object is in L2. On an L2DIR miss and L2DLAT hit, the L1-L2 transfers can proceed immediately if the target line is valid.

4.2 L2 Mapping

The L2 map is implemented as a two level hierarchy. The level one map resides in the L2DLAT. The level two map, the ***L2 page directory***, L2DIR, resides in the L2 store itself. Just as with a memory hierarchy, when the L2DLAT misses, the

L2DIR is accessed. Entries in the lower level of the map hierarchy contain less information than entries in the top level in an effort to keep the L2DLAT small, an implementation decision. Therefore a hit in the L2DLAT does not imply the L2DIR will not be accessed to complete the L2 mapping. This happens when L2DLAT.VL = 0 for the target line, indicating that although the L2A of the target is known, the line is not in L2. The L2DIR target entry is then accessed for the shared memory address.

The CPU must conceptually perform an L2 mapping (fig. 4.7 [1]) during each L1 mapping (fig. 4.6) to find the tag matching the L2A tag in the L1DIR. Usually the L2 map hits in the L2DLAT. Less frequently, the L2DIR must be accessed to complete the L2 mapping. Since the CPU will wait for both L1 and L2 misses to be resolved, L1 and L2 operations are sequential. The L2DLAT is considered part of the L1 system because it is physically partitioned with L1 to permit fast access. The hardware saved by keeping the L2DIR in the L2 store is considered worth the time penalty taken.

After a successful L2 map, L1 can still miss, but then L2DLAT.TAG will hold the target L2A, so no unnecessary work has been done. If L1 misses, the replacement line is written back to L2 if it is dirty. L1 also requests an L2 to L1 transfer of the target line. Both the upward and downward transfers can proceed concurrently if L1 has two line buffers each 16B wide. The 128B line transfers each entail eight 16B transfers. The inclusion of the line buffers adds one transfer time to the operation. The first line delivered to L1 is the target line. The first 16B transferred contains the

[1] Several algorithms in this thesis are outlined as flowcharts. Control follows directed branches. A split in a branch represents concurrent operations which must all complete at a merge point before flow continues. *Independent* branches, created at decision nodes, need *not* all complete at a merge point for flow to continue.

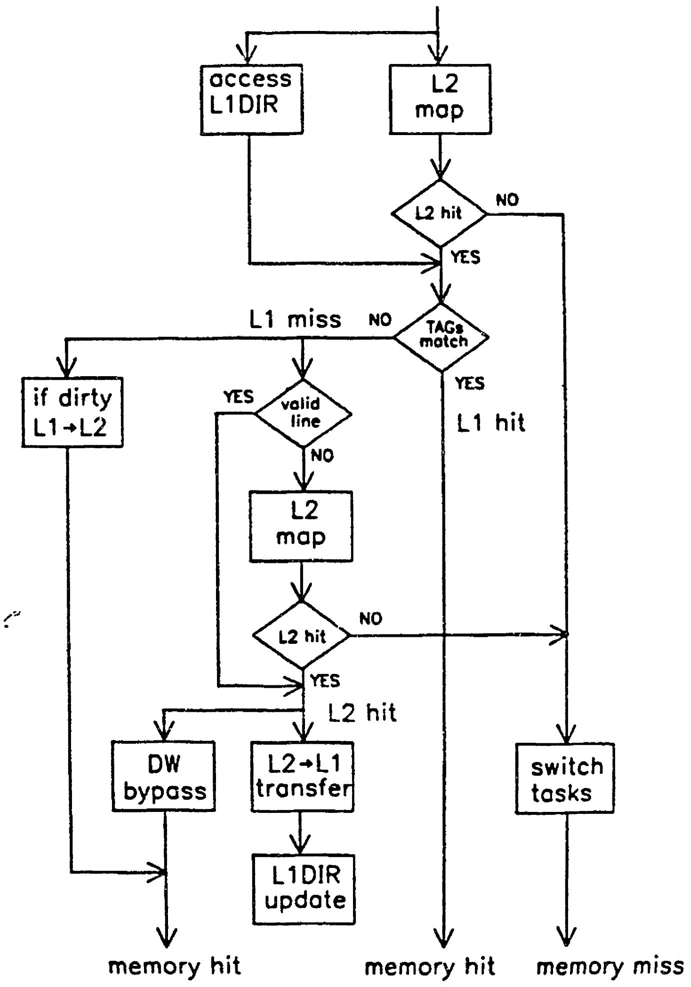


Figure 4.6. L1 Mapping

referenced double word and is routed to both L1 and the CPU. This is called the L1 *DW bypass.*

The L2 map can miss at both levels in the map hierarchy indicating that an L2 store miss. In this case, before the L1 mapping can be completed, an Ms mapping must be performed. L2 requests an Ms mapping and waits for the results. If the Ms mapping is successful, Ms signals L2 and sends it the target data object. An L2 page frame is selected for replacement and L1 is interrogated (section 4.3.1). L2 updates

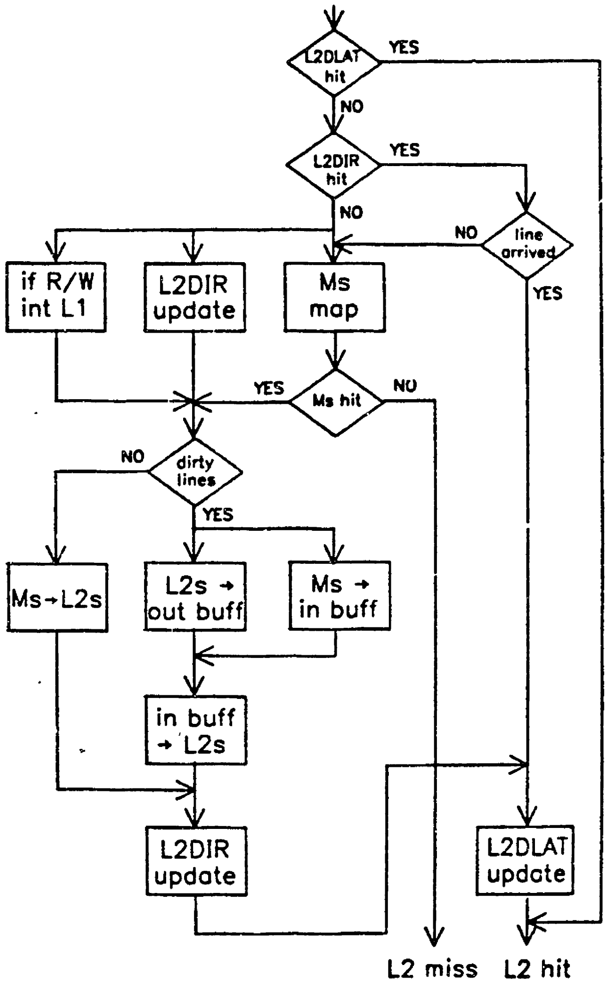


Figure 4.7. L2 Mapping

its maps with this information. It then proceeds as with an L2 hit. If the Ms map is unsuccessful, the Ms mapping function signals L2, L1 and the CPU and interrupts the supervisor. The supervisor will initiate a transaction switch on that processor. I/O will be initiated if the target Ms line is not already arriving. The precise details of the process switch will not be discussed.

L2 is demand paged over the whole L2 space, i.e. all pages, other than special pages such as map pages, are candidates for replacement. Therefore a running transaction can theoretically cause all other active transactions' working sets pages to be replaced except for those awaiting I/O. I/O pages are marked "arriving" and are effectively frozen.

The VA to L2A map is implemented with an "inverted" page table (HOUD81) because the size of this map is small, whereas the size of a conventional map would be much larger (because L2 can be viewed as a private main memory, extensive justification for the L2 map is given in section 4.4). A source virtual address is translated into an L2A by hashing it into a target address table (fig. 4.8). Two data structures needed are the ***hash index table,*** L2HIT, and the ***page frame directory,*** L2DIR. Both these tables are located in dedicated pages in the L2 store, L2s, requiring one and two pages respectively.

The L2HIT and L2DIR implement a chained hash table (HORO77). The source virtual page address is hashed to form an index into L2HIT. The hash function is based on the System/38 concern that sequential segment references should be prevented from affecting the uniformity of the hash (HOUD79).

$$hash(VA) = SID(11{:}0) + RVPA(11{:}0)$$

where RVPA is the reverse of the virtual page address. The VPA is reversed because both the SID and the VPA are likely to contain high order zeros if the virtual space is under-utilized and most files are small (POWE77). The hash is predicted to give a uniform spread of zeros. The L2HIT entry holds a pointer to the head link of the target hash chain in the L2DIR heap. The heap is of fixed size because there are a fixed number of page frames in L2. The lookup is implicit in the sense that the index of any entry in L2DIR is the frame number of the data object corresponding to that

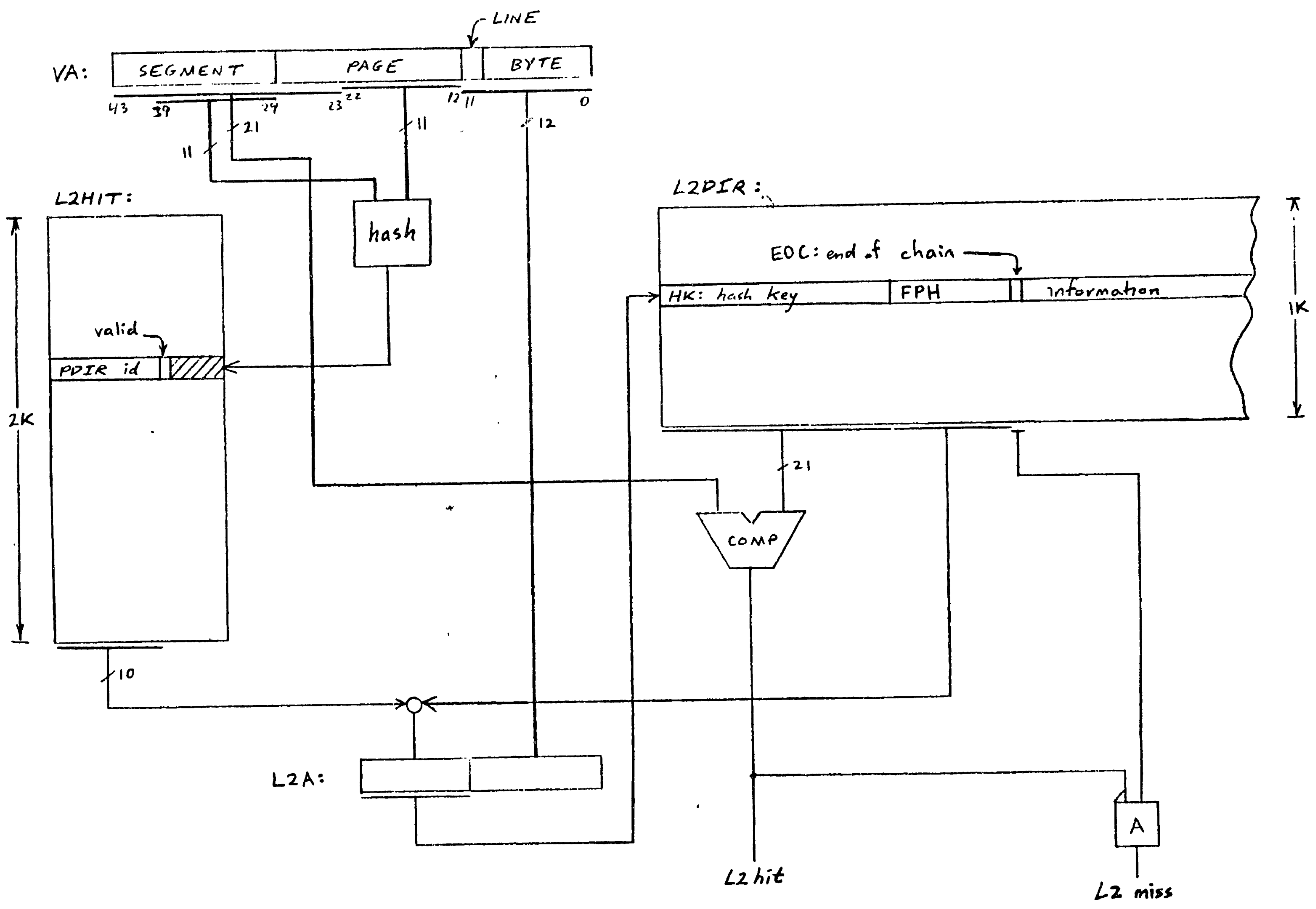


FIGURE 4.8. L2 Mapping Structures

entry. The entry table is defined to have twice as many entries as the heap for efficiency reasons. Essentially,

larger L2HIT -->
less dense hash structure -->
lower probability of collisions -->
shorter chains -->
faster mean access time.

Let a be the hash density defined as

$$a = \frac{|L2DIR|}{|L2HIT|}.$$

Then S_n, the mean number of accesses per lookup is given by (HORO77)

$$S_n = 1 + \frac{a}{2}.$$

Therefore if the L2HIT is twice as large as L2DIR, $a = 0.5$ and $S_n = 1.25$. S_n can be improved with a lower density, for instance two pages for L2HIT, giving $S_n = 1.125$. However, cost/performance as a function of L2HIT size can only be determined once the system is operational. The hardware mapping algorithm to be described works for any density with only minor modification in the obvious places.

An L2HIT entry [1] holds a pointer to L2DIR heap, PI, and a valid entry flag, VE. L2HIT entries not in use have VE = 0. Entries are invalidated when the single link comprising the hash chain is invalidated and the data object is possibly forced up the hierarchy. This occurs during transaction commit and L2 page replacement.

[1] Refer to L2HIT and L2DIR entry definitions in the L2c CARRIER declarations in Appendix B.

An L2DIR entry holds a hash key, HK, with which to determine a hit, a valid entry flag, VE, a foward hash chain pointer (to within L2DIR), FPH, and an end-of-chain flag, EOC. In addition, the entry holds information concerning the data object resident in the frame corresponding to the index of the entry. One link pointer is proposed, as is implemented in System/38 (HOUD81), to make the map update procedure efficient.

With a single link pointer, removing a chain entry at random, as in the case of replacement frame selection, involves scanning the complete chain (less the link to be removed) to reconnect the broken link. If the mean chain length is $1.25 \times 2 = 2.5$, plus another link for the hash index table entry, then 2.5 map accesses are needed on the average to cycle through a chain to the link behind the removed link. An additional access is needed to write that entry back. Thus the single link pointer removal penalty is $3.5 \times 50ns = 180ns$. The single link pointer insertion penalty for post-insertion is simply two pointer updates, entailing $4 \times 50ns = 200ns$ (single link pointer insertion penalty for pre-insertion is $6.5 \times 50ns = 330ns$).

For a map with backward and forward link pointers, these average delays are actually *longer*. The double link pointer removal penalty is two pointer updates (foward pointer of previous link and backward pointer of next link) or 200ns. The double link pointer insertion penalty is four pointer updates or 400ns. The advantage of the scheme is that the delays are constant for any length hash chain.

The difference between single and double link pointer removal plus insertion penalties 600ns - 180ns = 380ns. This is enough time to cycle through over seven extra links in the single pointer removal process. Since the probability of a chain of length 9.5 is very low, the single link pointer scheme is chosen.

The information fields in the L2DIR entry are VL, AL, DL, PK and MsA. The first three are the valid, arriving and dirty line bit vectors. PK is the protection key indicating a R/O, R/W or E/O data object. The MsA field holds the shared memory address of the parent data object. The advantage of the MsA is to obviate the need for an Ms mapping if the reference is a ***near-miss,*** i.e. if the Ms line is resident in L2, but the referenced L2 line is still invalid. However, if the Ms to L2 transfer size is 4KB, no near-misses will occur.

The L2 map is kept in the L2 store, which requires a 16B port to produce the necessary 1GBS bandwidth for bus transfers. Therefore two L2DIR entries of 8B each are produced on each access. If the hash is uniform, the probability of chaining into a sequential entry is very low and therefore the parallel access serves no advantage. However, the proposed hardware detects physically sequential links in the same chain to avoid unnecessary accesses if the hash is non-uniform.

There are three unique attributes of the L2 mapping policy. Most importantly, no transaction switch is taken on an L2 miss, i.e. the processor must wait for the Ms map and transfer. Secondly, an L2 miss immediately binds a replacement page to the target data object and writes back that L2 frame if the choice is dirty. This is done even if the reference misses in shared memory, causing I/O and a transaction switch. Third, the Ms mapping occurs concurrently with the L2 map update on an L2 miss. The update involves inserting the replacement frame L2DIR entry into the target chain and marking that entry arriving and frozen. Multiple I/O requests for the same object are thereby avoided.

The primary concerns in the design of the L2 mapping policy were simplicity, consistency and efficiency. By consistency it is meant that the actions taken on an L2 hit/miss should be the same for all transactions causing them. Specifically, when L2 misses, it immediately binds the current replacement frame choice to the target data object *before* the data object is delivered to L2, in fact possibly very much before that event. This policy does not adversely affect L2 hit ratio by much, considering that even with a high degree of multiprogramming, say 30 transactions per processor, there can be at most (30 / 1024) = 3% L2 pages arriving (recall L2 is demand paged only). This early binding saves overall map processing. Assume a transaction is switched in by the supervisor when its I/O completes. The transaction will begin execution by re-trying the reference which previously caused the I/O. This time, however, the L2 map will hit with the L2DIR entry marked "arriving", thus saving scanning the complete target hash chain (the arriving entry will be close to the head of the chain) and indicating that no identical I/O requests should be made by transactions sharing that data object. This policy is consistent because the original reference and retry are interpreted by the same hardware. Thus multiprogramming is transparent to the memory hierarchy. The mapping is efficient because an update is done during the initial search, rather than causing another search once the requested data object arrives.

The mapping and transfer hardware descriptions are given in Appendix B in the L2c (L2 controller) system, a part of L2. The organizational overview is given in fig. 4.9. The L2c dataflow is given in fig. 4.10. Figure 4.11 illustrates an example of the L2DIR update algorithm as described in the HEX **relink ROUTINE.** This figure shows a typical pointer structure of the replacement and target hash chains in the L2 map *before* and *after* the update triggered by an L2 miss. The update consists of removing the replacement choice link from the replacement chain and inserting it into the target chain.

4.3 L2 Page Fault Handler

Conventional page fault handlers are implemented in software and executed *after* a miss occurs. There are three major functions performed by a page fault handler: selection of a replacement page, the write-back of a dirty page and the downward transfer of the target page. The proposed L2 page fault handler is implemented in hardware and performs its selection function in *parallel* with the operation of the associated processor.

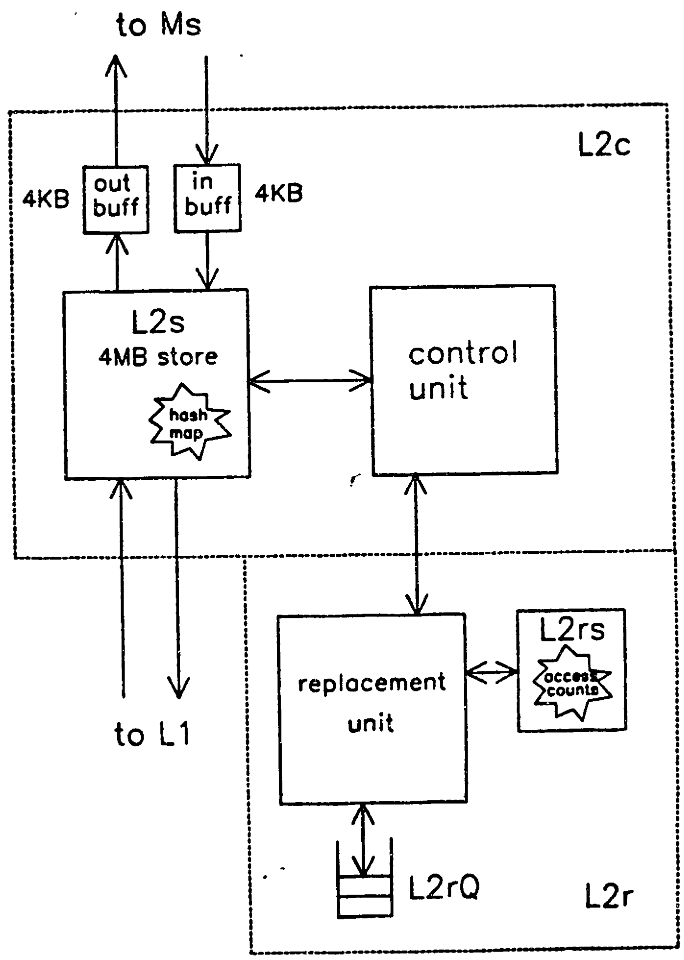


Figure 4.9. L2 overview

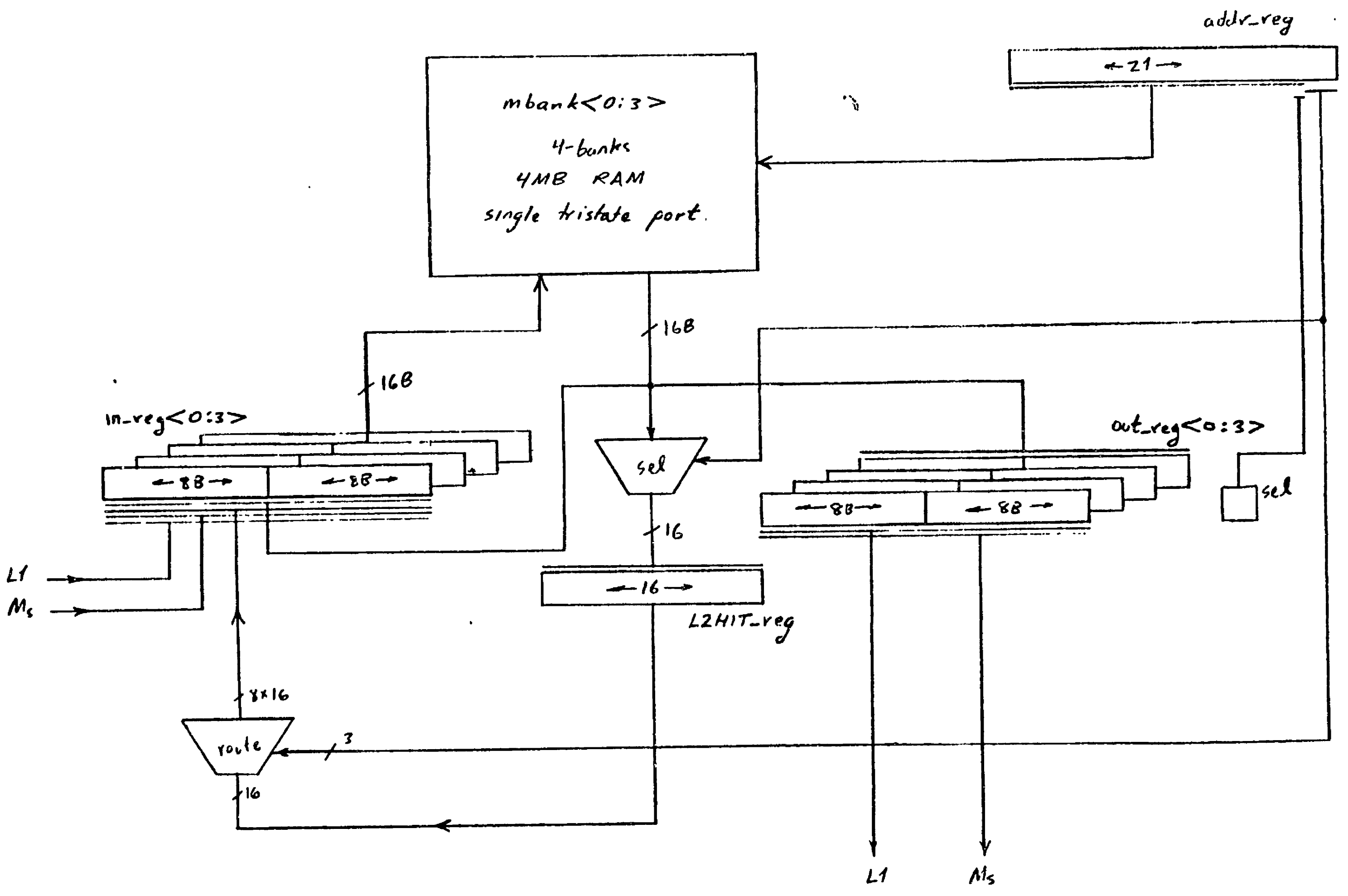


FIGURE 4.10. L2c Dataflow

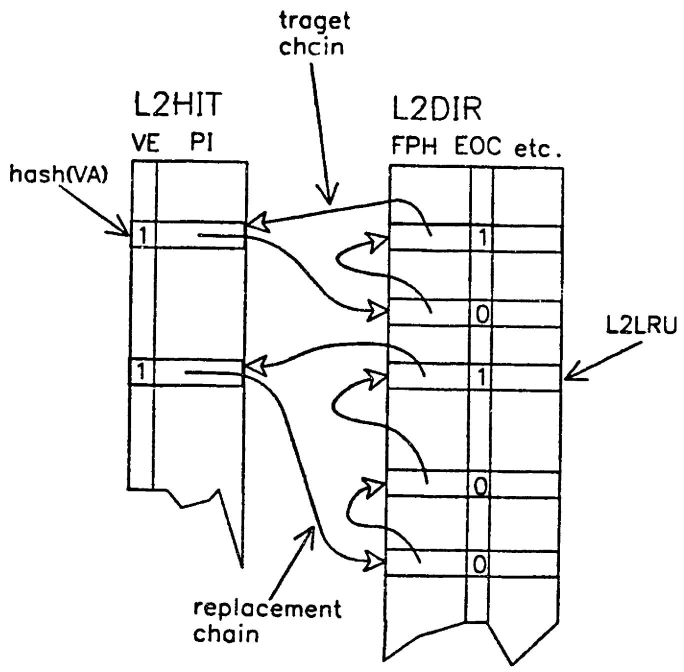


Figure 4.11a. L2 Map Update Example: Before

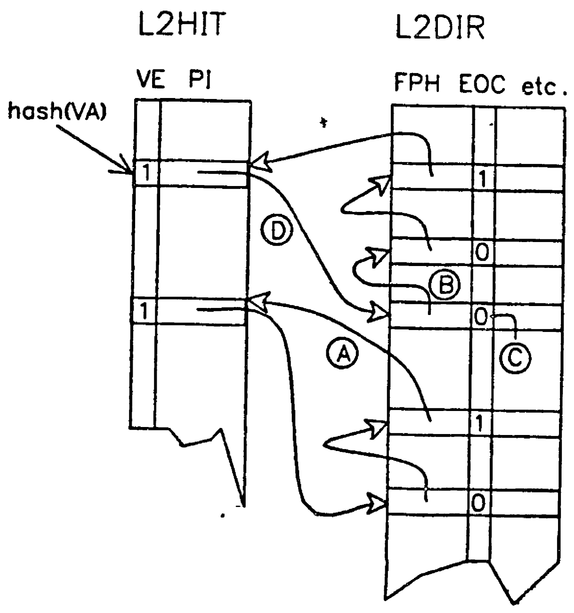


Figure 4.11b. L2 Map Update Example: After

4.3.1 Replacement

4.3.1.1 Algorithm

It is assumed in this thesis that the ***least recently used,*** LRU, criteria (EAST79) is a sufficient replacement algorithm for L2. Working set algorithms such as **Working Set**, WS (DENG78), **Damped Working Set**, DWS (SMIT76), and **Page Fault Frequency**, PFF (GUPT78) are not considered necessary at the L2 level because the L2 miss penalty is not as great as in a conventional system. No process switch is taken. Thus for replacement, L2 is viewed more like a cache than a private main store.

In a general memory hierarchy it is not possible to implement a page fault handler in software for each level. Such a scheme is highly inefficient. A conventional system usually uses a simple hardware line replacement mechanism for the cache and a software page replacement mechanism for the main memory. This is because whereas the time penalty of the main memory fault handler is insignificant compared to the I/O time, that same penalty is much greater than main memory access time. A second reason for using a software handler for the main memory is that the higher in the hierarchy, the more important it is to make the replacement choice intelligently. All lower levels, no matter how accurate their replacement algorithms, suffer by mistakes made higher up.

In the proposed system, L2 replacement must be implemented in hardware because of performance reasons. If the decision as to which of the L2 pages to replace is made after a page fault occurs the decision process must be on the order of 1us. This is roughly calculated from the assumed IFI_{mp} = 10us. An L2 miss, which causes the processor to wait, should not impact performance by more than approximately 5%. Therefore a simple and fast page fault handler is necessary.

An alternative approach is to select replacement pages concurrently with normal L2 operation by filling a replacement queue. In this case, the replacement hardware must fill the queue faster than the queue is emptied by L2 misses or invalidated by L2 accesses of enqueued choices. It is hard to estimate how often a new LRU choice need be produced, although it must average at least once every IFI_{up} = 80us. Worst case would be that each L2 access invalidates an LRU choice presently enqueued. That implies a new choice must be produced on average once every 8 us, to keep the queue filled in the steady state.

LRU approximation techniques, called ***bit scanning algorithms,*** are well documented (BAER80,HABE76,EAST79). For common fixed space algorithms such as CLOCK, TRIGGER SWEEP and SCHEDULED SWEEP, LRU accuracy levels off at only three bits per access count (EAST79). The algorithms presented here are two derivations of SCHEDULED SWEEP, called MINM and ZERO. Both use two bit access counts.

	MINM	ZERO
sweep: (periodically)	all counts <- 0	decr all counts
update: (on L2 access)	count <- timestamp	count <- '11 [1]
search: (on L2 miss or continually)	min(count)	count = '00

where the MINM timestamp is a modulo 4 counter, incremented four times between sweeps. No ***use bits*** are needed because the update directly modifies the access count. Each of the algorithms has its advantages and disadvantages. It is especially hard to

[1] HEX binary notation for 3.

compare their complexity because when using two bit access counts, they are similar. For short counts, the major delay in finding any zero in a set of counts is *prioritizing* the zeros that are found, i.e. hardware has just as must trouble finding any zero as the first zero. This delay is comparable to the selection delay in finding a minimum count because an arithmetic comparitor for two short fields requires few logic levels. For longer counts, ZERO algorithm wins because the prioritization delay is the same for all count lengths (complexity increases only with number of counts) but the minimization (and maximization) hardware become rapidly cumbersome and slow. ZERO is chosen arbitrarily.

The main idea in concurrent ZERO is to periodically ***sweep*** or decrement all the access counts. The ***search*** for a suitable replacement choice, i.e. any frame with a zero count, is performed continually. The current choice is put in a finite length queue. When a frame is accessed, its count is set to the maximum value, '11, and if that frame address is enqueued, that queue entry is invalidated. On a miss, a valid queue entry is taken as the replacement choice. If all the queue entries are invalid, a sweep and a search are triggered in order. The probability of the entire queue being invalid is *very low* if the queue size and the sweep rate are compatible. At the opposite extreme, the sweep must not be performed too often, or the LRU approximation breaks down. A trade-off must be made concerning the cost of invoking a sweep (no zeros) and the cost of a poor approximation (too many zeros).

The correct way to calculate the sweep rate and the replacement queue length is to use a trace driven simulation. The results will of course be related to the application on the trace. Because the proposed system will run widely varying applications, an accurate modelling effort would be time consuming. A worst case simulation was run assuming random L2 references with a Gaussian arrival time distribution. This showed about 15ms is the highest period not causing triggered sweeps on misses. Since L2 references with locality will increase the number of zero access counts, the sweep period can possibly be increased with no penalty. The effect of the finite queue length is not expected to lessen performance if the sweep period is adjusted accordingly. Rough calculations suggest that a 4 deep queue and 10ms sweep rate should trigger very few sweeps for random references. Again, locality of reference lessens this penalty.

A back-of-the-envelope estimation of the sweep period comes surprisingly close to the above *results*. A single processor L2 mean page fault interval is $IFI_{up} = 80us$. There are 1024 page frames in L2 and therefore the mean page life is $1024 \times IFI_{up} = 80ms$. To "synchronize" decrementing the access count with the page life, the sweep period should be 80ms / 4 = 20ms, because the access count is only two bits. This method of analysis is used for the estimation of the shared memory sweep period, implemented in a similar manner.

4.3.1.2 Implementation

The L2 replacement unit, L2r, consists of an access count and flag store, L2rs, a replacement choice queue, L2rQ, and a controller. The L2r dataflow is given in figure 4.12 and HEX description in Appendix B. The primary design goal of L2r is to keep L2rQ at least partially full. L2rQ is a four entry queue, each entry holding an L2 page frame address of a replacement choice, i.e. non-frozen frames with zero access counts. A bit vector, vQe, has a valid flag for each queue entry. If the flag is set, the entry holds a valid replacement choice. Causes for invalidation are access of the page frame after the choice was enqueued and entry use by L2c for actual replacement.

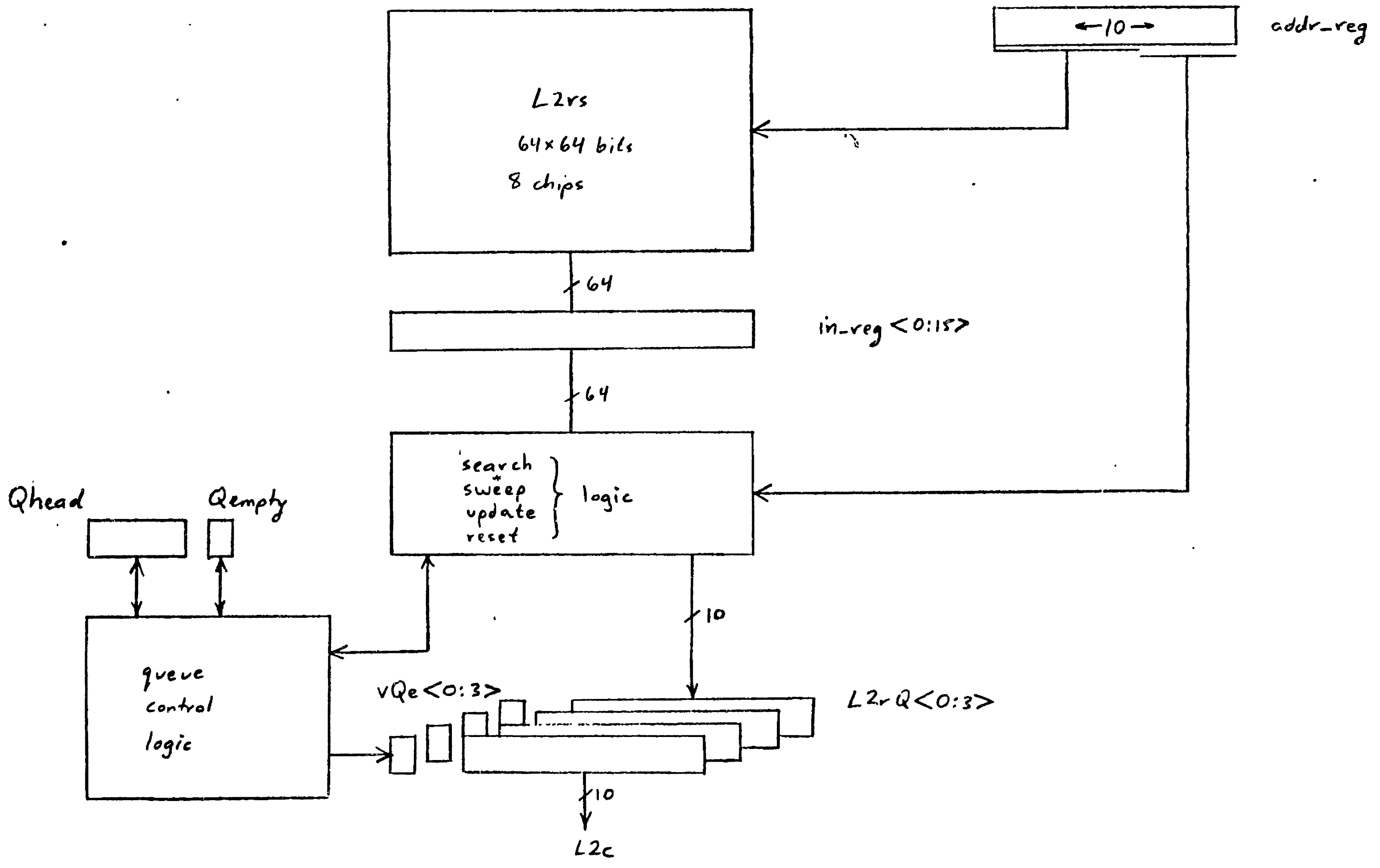


FIGURE 4.12. L2r dataflow

An L2rs entry holds the two bit access count and two flags, as indicated below.

```
L2rs entry
          bits
    AC     2       access count:
                   00 not ref.
                   01 ref. once
                   10 ref. twice
                   11 ref. > two times
    NR     1       never replace (permanent)
    DR     1       don't replace (temporary)
```

The two flags must reside in L2rs because they are needed in the replacement decision. Frames with either flag set cannot be replaced and the controller will not select them.

L2 will miss on the average once every IFI_{up} = 80us and assuming a 10% miss ratio, this implies an L2 access every 8us = IRI_{up}, the ***inter-reference interval.*** These constraints indicate that L2-technology memory can be used to implement L2rs. This is acceptable if several L2rs entries can be manipulated in parallel. A 16 entry parallel access L2rs is proposed, requiring 64 accesses to search the entire L2 store for replacement choices, taking 64 x 50ns = 3.2us. A sweep, invoked approximately every 10ms requires 128 accesses (read and write), taking 6.4us. This assumes the logic performing the search and sweep operations are hidden under the access time, which is the case. L2rs is therefore $64 \times 8B = 512B$, requiring eight 64 x 8 RAMs, plus parity. The controller and queue is partitioned onto three chips. Two chips contain the necessary logic to perform the update, search and sweep operations on 16 entries in parallel. Each operates on 8 entries. These chips are I/O pin limited at 32 inputs/32 outputs to L2rs and 10 to/from the controller chip. In other words, if the number of parallel entries was doubled, which is the only efficient increment in terms of addressing, a single chip would not be able to handle the I/O. The third chip is the master control chip and holds L2rQ, vQe and their corresponding update logic.

L2r operates in three modes corresponding to the phases of the ZERO algorithm: update, search and sweep. Update mode sets L2rs.AC to '11 and interrogates L2rQ when invoked an an L2 access. Update mode is also used to set and reset the flags. L2c will set L2rs.DR when requesting I/O for that frame. This is a temporary freeze. L2c will reset L2rs.DR when the first successful reference to a temporarily frozen page frame is made. This sequence is illustrated below.

L2DIR.VE		VL	AL	L2rs.NR	DR	comment
1.	0	XXXX	XXXX	0	X	pick replacement frame
	or 1	XXXX	XXXX	0	0	
2.	1	0000	0010	0	1	mark target line arriving mark all lines invalid freeze frame, map into Ms
3.	1	0010	0000	0	1	mark target line valid and arrived on Ms hit
4.	1	0010	0000	0	0	unfreeze frame on first ref

During processor cold start-up L2rs.NR will be set for page frames holding the L2 map and other special objects and reset for normal page frames. During local recovery procedures, a hardware mechanism will set L2rs.NR for all frames in a failed module. If the map itself has failed, the recovery manager will select an L2 area for the map and set L2rs.NR for those frames.

Search mode is the default or background mode of L2r operation. This means that the L2rQ is constantly being updated with recent replacement choices. The sweep mode period should be alterable to dynamically fine-tune the system.

4.3.2 Upward Transfer

The second major function of the page fault handling sequence is to force dirty pages to Ms when they are replaced. There is a problem however if the page selected for replacement has lines still resident in L1. If the L2 page is replaced, the L1 lines, if modified, have nowhere to be written back to. This is called the ***orphan problem.*** The upward transfer is therefore split into two operations: L1 interrogate and L2 write-back.

4.3.2.1 Orphans

The memory hierarchy presented can be characterized by memory levels of increasing size and page size.

level	name	page size	# pages
1	L1	128B	256-512
2	L2	4KB	1K
3	Ms	64KB	16K
4	Bs	64KB	256M

Any page with one or more copies resident in lower levels is called a ***parent.*** A copy is called a ***child.*** L1 lines and L2 pages are all children. L2 pages can be also be parents. By virtue of the store-in policy, a data object must migrate down through all the memory levels for both load and store references. Each level has *independent* mapping and replacement policies. The L1 replacement policy is a poor man's LRU among the sets within the target class. L2 replacement is a more sophisticated global LRU and Ms uses a complex software algorithm. The schemes are independent in the sense that it is possible for certain levels to replace a parent thus creating *orphans* in lower levels.

Orphans of R/O files are not a problem because replacement policy on each memory level simply invalidates R/O data objects without write-back. It is the R/W orphans that are trouble. The term orphan used in the remainder of this section refers to R/W orphans.

L2 parents can orphan their children because L1 is four-way set associative, causing lines to possibly reside longer than is warranted. Consider the following example involving a simple uniprogammed environment. The process stores into a DW, DW*, in a segment as yet unused. Through a sequence of faults, DW* is delivered to the CPU, leaving a parent, P*, in L2 and its child, L*, in L1. Assuming L1 is set associative with 64 classes, the mapping domain is also split into 64 mutually exclusive classes. Domain addresses in the same class share common hashes, i.e. low order bits. One sufficient condition for the creation of orphans is if no other references made by the transaction fall into the DW* class. An example of this is if the transaction never again accesses the first 128B in all referenced virtual pages, after initially accessing L*, the first 128B of some page. L* will then never be replaced from L1, yet P* will eventually be chosen for L2 replacement because L2 is fully associative. Note that P* has no dirty 1KB lines, *yet it is not up-to-date.*

A solution is to ***interrogate*** L1 on L2 replacement. The interrogate is performed using the L1DIR and L2DLAT in the same manner in which the CPU references L1. This is possible because the CPU will wait during an L2 page fault and free up L1. The interrogate can be done for either all valid 128B lines in a 4KB L2 page or only those L1 lines known to have been transferred to L1. The second alternative is costly - a 32 bit multiple copies vector must be added to the L2rs entry, increasing L2rs size by 800%. The bit corresponding to a 128B line in an L2 page would be set when that line was transferred to L1. The bit would be reset on write-back. Since clean lines would never be written back, the multiple copies vector would be a worst case indicator. In fact, if this vector was implemented, a possible solution would be to simply never replace L2 parents. This would require only a 6 bit multiple copies tally, increasing L2rs size by only 150%. The tally would be incremented when a line was transferred to L1 and decremented when a line was written back. A non-zero tally indicates a potential L2 parent but not *who* its children are. Even in the absolute worst case of 512 different L2 pages each having an only child, there would still be 512 L2 pages without children. Realistically the number of parents would be lower because of locality of reference. The success of this solution hinges on the fact that the longer an orphan resides in L1, the higher the probability it will be replaced shortly. In the time between L2 and Ms faults of the same family, the L1 orphan may be naturally replaced. Thus work is saved. A disadvantage is that the L2 hit ratio may be lowered because the replacement set is smaller.

The first alternative of brute force interrogation on each L2 replacement does not increase the L1 miss penalty by much because the interrogate time is hidden under the Ms to L2 transfer time, as will be shown. It also involves less hardware than the other scheme, and is therefore selected for the L2 replacement protocol.

4.3.2.2 L2 Write-back

To alleviate the predicted contention for Ms one possible scheme is to do "sneaky writes" attempting to keep all pages in L2 clean. A general problem with sneaky writes is that they create possibly *extra* Ms contention because they are not completely efficient. This inefficiency arises from the chance that the cleaned pages will be dirtied again *before* the next page replacement. In the BIP MP, sneaky writes can also cause performance degradation by contending with the CPU for L1, to do interrogates. A simple write-back policy is proposed, entailing writing back a replaced page if it is dirty, once chosen for replacement.

To reduce the bottleneck caused by the write-back, two buffers assist in the L2 to Ms transfer. Both **in_buf** and **out_buf** (Appendix B) are 4KB stores implemented in L2-technology. They are 4-way interleaved to accept 1GBS transfer rates. Each consists of four 64 × 16B banks, built with 64 × 8 RAMs. **out_buf** is logically partitioned into 32 L1 lines and is directly line addressable. OBV(31:0) indicates whether an **out_buf** line is valid. **in_buf** is logically partitioned into four 1KB L2 lines. IBV(3:0) indicates whether an **in_buf** line is valid. There is an **in_buf** bypass from the bus for direct Ms to L2 transfer if there is no write-back.

The timing of the L1 miss penalty, a substantial portion of which is contributed by the L2 page fault handler, is given in fig. 4.13. The rough timing of these operations are given below.

L2 mapping = (1 + 1.25) x 50ns = 100ns
L2 map update = 10 x 50 ns = 500ns
L1 interrogate = (32 + 2) x 20ns = 700ns if 0 orphans
700ns + 8 x 10ns = 0.8us if 1 orphan
⋮
700ns + 32 x 80ns = 4.0us if 32 orphans
L2 write-back = 0us if 0 lines
8 x 130ns = 1us if 8 lines
⋮
32 x 130ns = 4us if 32 lines
Ms mapping = 2.25 x 50ns = 100ns
Ms to in_buff transfer = 1us + 1us = 2us if 1KB
⋮
1us + 4us = 5us if 4KB
in_buff to L2 transfer = 1us if 1KB
⋮
4us if 4KB
L2 map update = 2 x 50ns + 20ns = 120ns
L1 to L2 transfer = 130ns
L2 to L1 transfer = 130ns
L1DIR update = 20ns

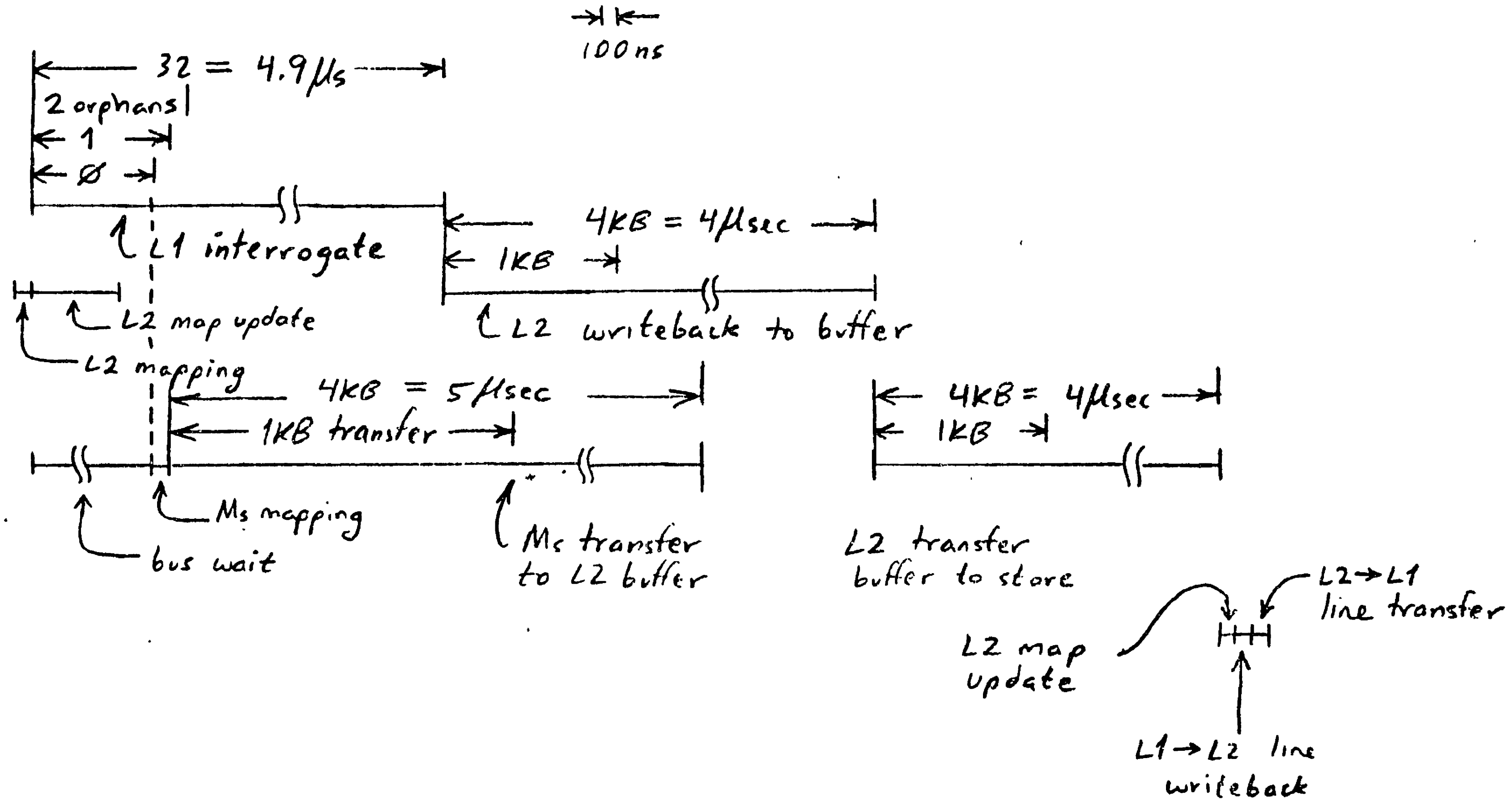


FIGURE 4.13. L1 miss penalty

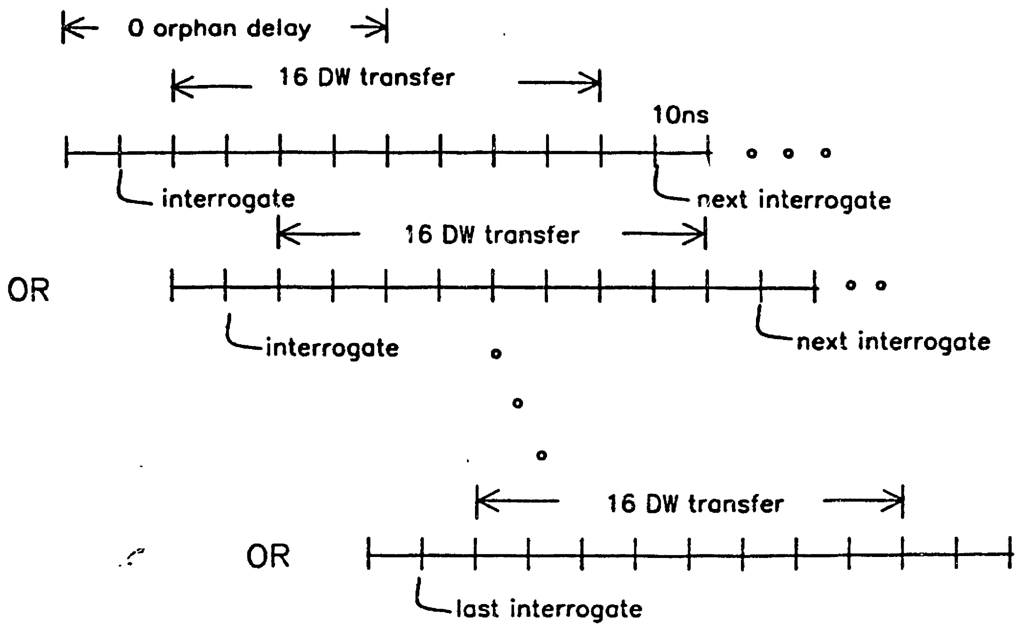


Figure 4.14. L1 Interrogate Timing

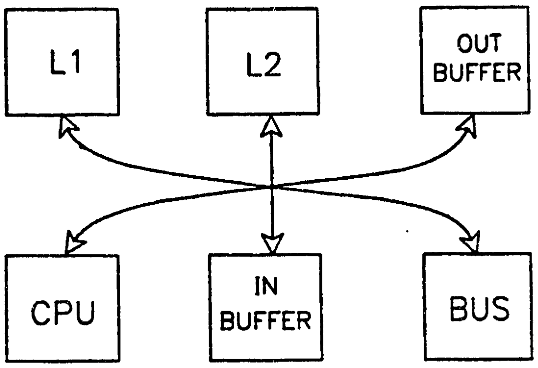
The first L2 map update involves inserting the replacement frame into the target hash chain. The second L2 map update involves marking the arriving line as valid and arrived. The first and second terms in this operation correspond to the L2DIR and L2DLAT updates respectively. The L1 interrogate delay for a given number of modified orphans consists of a transfer delay and a constant mapping delay (fig 4.14). The transfer delay is the time necessary to transfer the 128B orphan line to the L2 buffer in eight 16B transfers. The mapping delay is taken even if there are no orphans. This delay consists primarily of two L1DIR accesses for each of 32 lines interrogated: one read and one write (to invalidate the entry). The additional four accesses correspond to loading and then invalidating the L2DLAT with the replacement frame L2A, usually necessary since that page is likely to be unused.

The expected frequency of orphan creation in any application will be very low. Assume that one orphan is created by each L2 replacement and that there is no bus contention. With write-back of four L2 lines, the upper timing sequence is about 5us. If the Ms transfer is the maximum 4KB taking 5us, the buffering saves about 4us of bus usage and the total L2 miss penalty is about 10us. If the Ms transfer is 1KB taking 2us, the miss penalty is about 6us. The buffering not only saves 4us bus usage, but also reduces the miss penalty by 1us. If the write-back is only 1KB, the upper sequence is about 2us. Now if the Ms transfer is 4KB, the buffering saves only 1us of bus usage at the cost of an additional 3us in miss penalty. In this case the miss penalty is 10us instead of 7us without buffering. If the Ms transfer is only 1KB, the buffering saves 1us of bus usage with no increase in miss penalty. If there is no write-back, buffering is bypassed and the miss penalty is about 6us and 3us for 4KB and 1KB transfers respectively. The CPU, L1 and L2 inter-connections are summarized in figure 4.15.

4.4 Ms Mapping

This section describes the shared memory mapping algorithm and its implementation. The proposed scheme is an "inverted" page table lookup implemented in high-speed (L2-technology) memory. This method is similar to the L2 mapping and will be justified by consideration of the consequences of a conventional segment/page table mapping, e.g. MULTICS (BENS72).

The scheme, described in detail in section 4.2, uses an "inverted" page table map so-called because its size is proportional to the size of the target space, not the source space. That is the fundamental advantage of using a hash table. A conventional translation uses a map whose size is roughly proportional to the size of the virtual



CONNECTION	COMMENT
1. CPU → L1	processor reference
2. L1 → L2	L1-L2 transfers
3. L1 → OUT BUFF	L1 interrogate
4. L2 → OUT BUFF	L2 write-back
5. BUS → OUT BUFF	L2 → Ms transfer
6. BUS → IN BUFF	Ms → L2 transfer
7. BUS → CPU	Ms map reference
8. BUS → L2	buffer bypass
9. L2 → CPU	DW bypass
10. L2 → IN BUFF	Ms → L2 transfer

Figure 4.15. Private Memory Connections

address space because the map is indexed with the VA. This was the main motivation for using hash tables, especially since the virtual address space is large. The 64KB virtual page size, defined to be equal to the Ms page size, was also chosen to reduce the size of the shared memory map because Ms frame size is inversely proportional to number of frames.

4.4.1 Conventional Method

A conventional UP, e.g. IBM 370/3033, main memory map is invoked when the DLAT misses. The map segment is stored in the memory hierarchy and is, in the simplest case, demand paged like all other segments. To increase translation efficiency, all or pieces of the map are frozen in main memory. The translation is performed by a hardware mechanism integrated with the CPU which accesses the map, possibly causing main memory faults, to generate the target address.

The location of the hardware translation mechanism must be decided for an MP system. Each processor could be capable of VA to MsA translation, either at the CPU level or at the L2 level (in L2c), or a single translator, attached to the shared memory, could be supplied. It will be shown that neither design is successful for a conventional mapping.

The possibility of 32K segments, each 8K pages (64KB per page), might preclude fitting either the segment table (descriptor segment) or the page table (information segment) in a single page. The severity of these sizes depends on the amount of information stored in a table's entry. As will be shown, the information segment fits in a single page, but the descriptor segment does not and therefore a descriptor segment page table is needed. Assume that three accesses are needed to map a VA to MsA (fig. 4.16). This mapping is:

$$\text{MsA(target)} \leftarrow \text{IS}\ (\text{DS}\ (\text{DSPT}\ ((\text{DBR}) + \text{SID}') + \text{SID}'') + \text{PID}) + \text{W}$$

where SID', SID" and W are fields within the segment identifier and word index of the VA, respectively. PID is the page index of the VA. DBR is the descriptor base register, DSPT is the descriptor segment page table, DS is the descriptor segment and IS is the information segment [1]. SID' and SID" can be defined only once the DS entry size has been determined. The DBR is needed to possibly relocate the DSPT in Ms in case of memory failure.

[1] Much of the notation used in this section is borrowed from (BENS72).

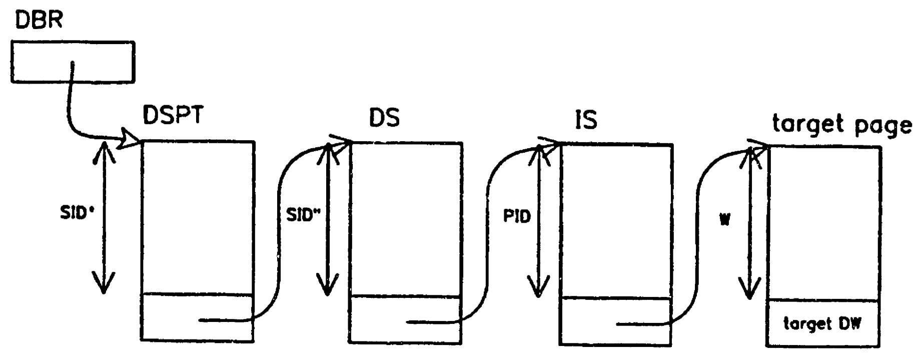


Figure 4.16. Conventional Main Memory Mapping

The map entries for the DSPT, DS and IS are given below. Recall there are 2^{14} page frames in Ms.

		bits	
DSPT entry:	DSP	14	pointer to DS in Ms
	VDS	1	DS valid flag
	ADS	1	DS arriving flag
DS entry:	ISP	14	pointer to IS in Ms
	VIS	1	IS valid flag
	AIS	1	IS arriving flag
	PK	3	protection keys
	ST	2	segment type
IS entry:	PP	14	pointer to page in Ms
	VL	16	valid 4KB line
	AL	16	arriving 4KB line
	DL	64	dirty 1KB line

These entries have been formulated to support only information crucial to VA to MsA translation. All information concerning translation from symbolic file name to virtual address to physical disk address is kept in another map, called the **segment descriptor table,** SDT (section 5.3.1). The VA to MsA mapping misses if either the DSPT, DS or IS entries indicate that DS, IS or the target page is invalid or arriving. These misses invoke the Ms page fault handler.

The sizes of the DSPT, DS and IS maps are calculated as follows, where segno is the number of in-use segments.

$$|DS| = 2^{15} seg \times (|DS\ entry| = 4B) = 128\ KB$$

$$|DSPT| = \frac{|DS|}{64KB} \times (|DSPT\ entry| = 2B) = 4\ B$$

$$|IS| = segno \times 2^{13} pg/seg \times (|IS\ entry| = 16B/pg)$$

$$= segno \times 128KB$$

Conventional mapping therefore requires a map proportional in size to the number of in-use segments. Assume that the degree of multiprogramming is roughly 200 and each active process uses five segments, i.e. segno = 1000. This assumption is compatible with a file space of 1% of the virtual space, although the in-use *space* has no bearing on the conventional map size. Therefore

$$|map| \cong |IS| = segno \times 128KB = 128\ MB$$

In an effort to speed up translation, the complete DS is frozen in shared memory, thereby reducing the number of accesses per translation to two. The DBR then points to the head of the DS. The new mapping is:

MsA(target) <-- IS (DS ((DBR) + SID) + PID) + W

There are two choices for managing the map: demand paging like other segments or freezing the accessed IS pages for the life of the involved transaction. On transaction commit, the system unfreezes associated IS pages and possibly marks them for replacement. Although the map is big, for a realistic active file space, the worst case policy of freezing the map uses only 14% of Ms.

The "inverted" page table mapping uses hash tables of the size (entry definitions in section 4.4.2):

$$|map| \propto |Ms|$$

$$|MsHIT| = 2^{14} \times 2 \times (|MsHIT\ entry| = 2B) = 64\ KB$$

$$|MsDIR| = 2^{14} \times (|MsDIR\ entry| = 22B) = 352\ KB$$

The hash map is over 300 times smaller than the conventional map.

The major point against the conventional map is that most segments will not be 8K pages long and thus a great percentage of IS space will be wasted. This degrades processor performance because a significant percentage of the private processor memories will be filled with IS pages, leaving less room for working sets, increasing the private memory miss ratio. Another point against IS pages is that they could be kept in a portion of Ms implemented in L2-technology *if they weren't so numerous.* Such a map store, logically part of Ms, could be accessed by a hardware translation mechanism at the shared memory level. IS pages would never migrate below the shared memory and the private memory contention problem would be solved. The map store can, however, be cost effectively implemented for a hash scheme because it uses a smaller map.

The choice of an "inverted" page table for VA to L2A translation can now be justified. For L2, the conventional map needed per processor is approximately 16 times bigger than the VA to MsA map because L2 page size is 1/16 Ms page size. The hash map size is calculated below.

$$|map| \propto |L2|$$

$$|L2HIT| = 2^{10} \times 2 \times (|L2HIT\ entry| = 2B) = 4\ KB$$

$$|L2DIR| = 2^{10} \times (|L2DIR\ entry| = 8B) = 8\ KB$$

That makes the conventional map over 150,000 times larger than the hash tables used! Realistically, a slightly more intelligent design could reduce the conventional map size, for expected file size distributions, to roughly the Ms map size. Even then, the map is 32 times as large as the L2 store itself.

4.4.2 Proposed Map

The hash entry table, MsHIT, and the Ms page directory, MsDIR, are defined as follows.

MsHIT entry:

	bits	
PI	14	pointer to MsDIR heap
VE	1	valid entry flag

MsDIR entry:

HK	17	hash key
FPH	15	hash chain link pointer
EOC	1	end of chain
PL	14	pool chain pointer
VL	16	valid 4KB line
AL	16	arriving 4KB line
DL	64	dirty 1KB line
MC	16	multiple copies
PK	3	protection key
ST	2	segment type
IOF	1	I/O buffer frame
SPG	1	shared page
REF	1	first reference
VE	1	valid MsDIR entry
FF	1	frame frozen
BAD	1	bad frame (failed memory)

The MsDIR.BAD, DL, VL, AL, REF and MC fields are the only sections of the map updated by hardware mapping unit, referred to as the shared memory controller, MsC. The fields are updated by the MsC as follows (reference to a bit vector implies the bit position corresponding to the target line).

- BAD is set by a hard error detection device in MsC.
- DL is set corresponding to the write-back from L2 of a modified 1KB line. This vector must be 64 bits to reduce the log granularity (section 10.3) to 1KB.
- VL is set when a line successfully arrives from disk by channel device transfer.

- REF is reset when a line successfully arrives from disk by channel device transfer. REF is set when the next reference to that page is made, i.e. whenever REF = 0. REF is used to generate a psuedo-fault interrupt (section 7.3).

- AL is reset when an I/O buffer line is successfully referenced. A 16 bit AL vector as opposed to a single bit flag, is needed to avoid requesting the same I/O.

- MC is set corresponding to the transfer of a data object from shared memory to a private memory. If this transfer is less than 4KB, the MC bit corresponding to the Ms line containing the data object is set. If the data object is R/O the MC bit is still set because the COMMIT and ABORT procedures need to invalidate *all* copies of data objects brought into memory by a transaction, even if they have not been modified. A 16 bit MC vector is chosen, instead of a 5 bit tally, to reduce the interrogate penalty for these procedures. The processors can be interrogated for 4KB objects using the INTL instruction because MsDIR.MC gives all child addresses. If only a tally was kept, the INTP instruction would be needed.

There are two reasonable implementations of the algorithm:

	map location	**MsC location**
Distributed :	Ms and L2 stores	L2c
Centralized :	dedicated store	Ms level

In the first scheme the Ms map migrates to the processor private memories, but not beyond the L2 level, where the translatión occurs. In the second scheme the map is fixed in a high-speed map store at the shared memory level.

Justification for keeping the map in an L2-technology store in both schemes is as follows. The L2 miss service time s, for the Ms map implemented in Ms-technology is

$$s = mapping + latancy + transfer$$

$$= 2.25us + 1.0us + 1.0us = 4.25\ us$$

where a shared memory packaged access time of 1us and transfer size of 1KB are assumed. The mapping time is calculated as 2.25 Ms accesses. Additional time is required to select the map entry from the 1KB staged in the Ms output registers. If the next link in the hash chain is in that 1KB, however, the next access is avoided. Since there are 256KB in MsDIR, if the hash was uniform then the probability of the next link landing in the current 1KB is very small. To account for other processors contending for Ms [1],

$$w = \frac{s}{1-\frac{s}{k}} = 4.\frac{25}{1-4.\frac{25}{10}} = 7.4\ us$$

where k = IFI_{mp} = 10 us. In time w the processor involved could have executed

$$\frac{1\ BIPS}{8} \times 7.4\ us = 925\ instructions$$

and since a single processor misses once every IFI_{up} = 80us, the penalty taken by the processor is

$$\frac{7.4\ us}{80\ us} = 9\%.$$

This assumes a hash table density of a=.5, which is costly in that the hash index table must be twice the size of the heap. If a density of a=1 is used, the expected number of accesses goes up:

$$w = \frac{5}{1-\frac{5}{10}} = 10\ us$$

$$\frac{1\ BIPS}{8} \times 10\ us = 1250\ instructions$$

$$\frac{10\ us}{80\ us} = 13\%\ penalty$$

[1] Refer to section 1.1.2.

Neither result can be tolerated. Notice however that the number of instructions missed is too small to justify a task switch because most operating systems require more than 2,000 instructions to perform the switch (COCK81). Local hardware implemented switching is a possibility, but will not be considered here. The single processor penalty is further degraded by other Ms contention - map updates on an Ms miss and I/O traffic. These effects can be calculated using a queueing model but will certainly not improve the previous result.

The mean service time of VA to MsA translation when the map is accessed from an L2-technology store is

$$s = mapping + latancy + transfer$$

$$= 0.11us + 1.0us + 1.0us = 2.1\ us$$

$$w = 2.\frac{1}{1-\frac{2.1}{10}} = 2.7\ us$$

$$\frac{2.7\ us}{80\ us} = 3\%\ penalty$$

For conventional map needing only two accesses, the penalty is less than 0.1% better and therefore is not worth the cost.

An advantage in integrating the MsC and L2c is that the dataflow is similar for both L2 and Ms mappings, so that cost is reduced. MsDIR and L2DIR entry fields do not match perfectly, however, so that a combined function L2c will require additional hardware (the design *complexity* will not increase). A disadvantage of the centralized map is that it is vulnerable to failures. To increase its reliability, multiple copies of the store would have to be kept, increasing its cost. The distributed map does not have this problem because if the Ms module containing the map fails, the map is simply initiated in another module. A disadvantage of the distributed map, however, is that multiple copies of the same Ms map will exist on the private memory level. This infringes on working set space.

In addition a consistency problem exists between map updates made by the supervisor during Ms page fault handling, committing and aborting, and the processors during Ms mapping (marking dirty bits, multiple copy bits, etc). These concurrent updates will cause an inconsistent Ms map unless all processors but one are locked out while an update to the central map copy is made. This causes disastrous contention and so only the centralized map scheme is considered. The centralized map is simple and efficient. Although it is expensive because two or three copies of the map must be kept at the Ms level, it is still comparable in a sense to the decentralized scheme where eight copies of the map are kept at the L2 level.

The central map store is not logically part of the shared memory, permitting a tailored fit to the entry size of MsHIT and MsDIR entries, approximately 2B and 21B respectively. MsHIT holds 32K entries requiring $2 \times 32 = 64$ L2-technology chips ($1K \times 8$). MsDIR holds 16K entries requiring $21 \times 16 = 336$ chips. Error correction and detection bits must be added to these sums. There is an additional lock bit proposed for each MsHIT and MsDIR entry. Since only one map entry is accessed at a time, these bits can be kept on a single chip. The lock bits are set and reset by instructions described in section 3. If the MsC accesses a map entry which is locked, it waits for the lock to be reset and then reaccesses the entry. The MsC will interrupt the supervisor if a timeout condition is reached and the entry is still locked. Similarly, the MsC will lock an unlocked entry before it updates it, and then unlock it. These bits prevent the MsC from modifying an entry that the supervisor is concurrently changing and visa-versa.

The Ms store will have to be backed-up for reliability. This must be in a similar technology because of the frequency of accesses. Simultaneous updates of all copies is suggested. The MsC is similar to L2c; however, there is no hardware replacement unit corresponding to L2r because replacement is done is software.

To implement the architected map access instructions (section 3), a special bypass of the private memory must be made on Ms map references. In other words, 16B referenced from the map segment must be delivered from the Ms map store over the system bus directly to the CPU. Since the proposed MsDIR entry is 21B, two transfers will be needed per entry. This will take approximately 100ns.

4.5 Ms Page Fault Handler

In the normal course of its operation, the Ms mapping will miss, i.e. an MsDIR hash chain will be unsuccessfully scanned for a target virtual address. In this event, the MsC interrupts the supervisor. It sends the virtual address of missed data object and indicates whether the fault is ***real*** or ***psuedo.*** A real fault refers to demand paging and is described in this section. A psuedo-fault refers to an artificial mechanism used to pre-page a sequential file (section 7.3). The Ms page fault sequence is summarized in fig. 4.17. The ***Ms page fault handler,*** a member of the global operating system, is invoked on the interrupt. The Ms page fault handler performs a serial replacement algorithm, i.e. it makes the decision which page to replace *after* a page fault. A global fixed size LRU management policy is used. This refers to fact that the handler keeps the page frame allocation of a single transaction fixed during replacement. The LRU criteria is used and approximated in a similar manner to L2r - with 2 bit access counts.

Allocations are globally altered by the ***resource allocation manager*** by changing the degree of multiprogramming based on the real time aggregate measure of IFI_{mp}. Global allocation issues will be briefly discussed in the context of the replacement algorithm.

Even though MsDIR is kept in a high-speed store, a problem arises if the Ms page frame access counts are kept in each table entry. As was shown, LRU approximation algorithms which select a suitable page for replacement by comparing page access counts, require a periodic sweep mechanism which uniformly operates on the complete

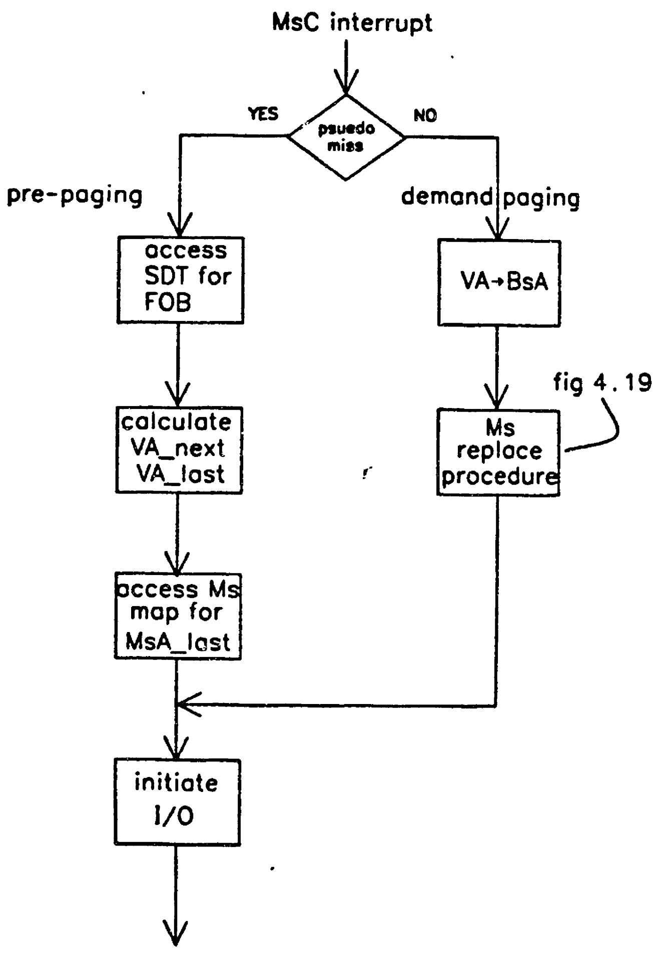


Figure 4.17. Ms Page Fault Sequence

set of access counts. For algorithmic efficiency, it doesn't matter what the operation is (resetting, incrementing or decrementing are common) if several counts can be processed in parallel during this stage. MsDIR, however, is a 16K serial entry table because each entry is too long to permit a parallel access implementation. Therefore if the count was kept in MsDIR, even an algorithm using the fastest sweep operation, reset, would require 16K MsDIR write accesses at 50ns/access = 800us. Such a delay cannot be tolerated, especially since it would have to be invoked at a high enough rate to successfully approximate an LRU algorithm.

The proposed organization is to keep groups of 64 access counts in the Ms map store, requiring 256 extra (partially empty) entries. The Ms map store overview is shown in figure 4.18. These groups are processed concurrently using the ZERO algorithm (section 4.3.1.1). The sweep will therefore require approximately 26us, sufficiently fast considering a sweep period of about 16ms. Sweep period is estimated as outlined in section 4.3.1.2. The Ms page fault interval is IFI_{mp} / mr = 100us, where mr is the shared memory miss ratio, assumed to be 0.1. Assume that the degree of multiprogramming is 210 and that each active process is given about 80 page frames, i.e. an equal share of Ms. There can be at most eight running processes at a given time, so that Ms page faults are effectively generated from $8 \times 80 = 640$ page frames. The mean page life is therefore $100us \times 640 = 64ms$ and the sweep period is 64ms / 4 = 16ms for two bit access counts.

The access counts can be accessed by the supervisor with the same machine instructions used for referencing the Ms map (section 3). The MsC updates the access count of a referenced Ms page frame by setting it to '11. The MsC also performs the periodic sweep. The Ms page fault handler only reads the access counts at the beginning of the replacement algorithm. The lock bit associated with each 64 access

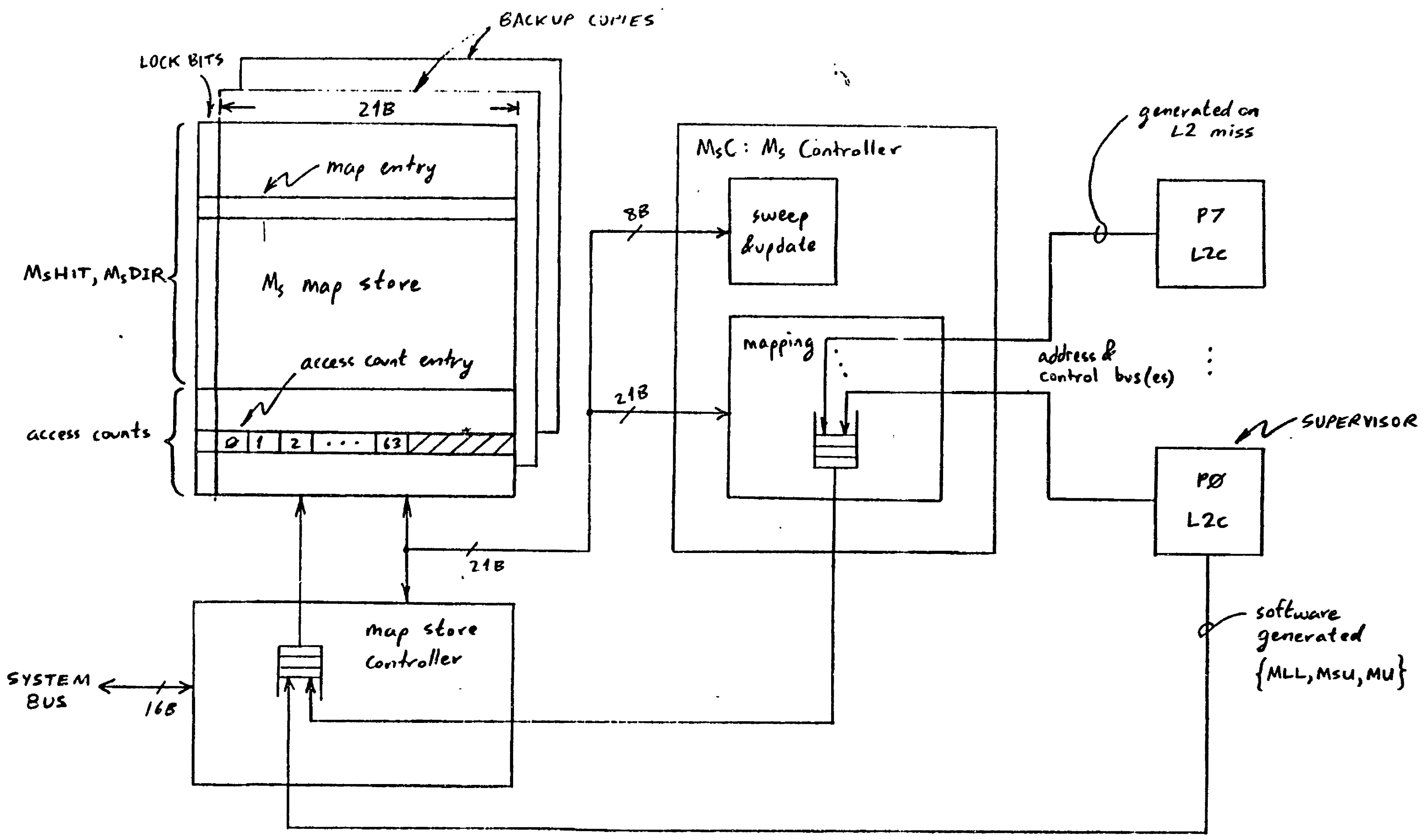


FIGURE 4.18. M_s Map Store Overview

count entry is not needed because the Ms page fault handler locks the map entry corresponding to an access count *before* it reads the count [1]. Thus any attempted mapping into a frame considered for replacement will be blocked, i.e. the access count will remain constant. The replacement frame is eventually chosen and its map entry is updated and frozen. At the end of the replacement algorithm all involved map entries are unlocked. The sweep must be disabled during Ms page replacement to prevent the supervisor software from seeing an inconsistent set of access counts.

The replacement policies concerning processes running and not running under level-3 consistency are different. For processes not guaranteed consistency, dirty Ms pages are written back to secondary memory on replacement. This conventional policy will not be discussed further.

Modifications made by transactions running under level-3 consistency are not permitted to be written back to secondary memory, a constraint of the locking protocol (section 8). This has the additional benefit of reducing system bus contention. All dirty pages are effectively frozen in Ms until all lock holder transactions commit. The COMMIT procedure forces the modified pages to Ms (section 9.1). This policy is a uncommon approach to replacement, afforded by the extremely large shared memory. It is also the shared memory manager's policy to effectively freeze all pages doing I/O, called ***I/O buffer frames.*** An I/O buffer frame doing input is unfrozen when the I/O transfer is complete *and* a reference is made to that data object, either by the transaction causing the I/O, or by another transaction sharing the object. This is because the buffer frame should not be chosen for replacement before the data object can be successfully used.

[1] Half way point! Don't give up, you're almost finished.

The basic steps in this input scheme are summarized below.

MsDIR.VE		BAD	FF	VL	AL	comment
1.	0	0	X	XX...XX	XX...XX	pick replacement frame
or	1	0	0	XX...XX	XX...XX	
2.	1	0	X	00...00	00..010	mark trgt line arriving mark all lines invalid initiate I/O
3.	1	0	X	00..010	00..010	mark trgt line valid when I/O completes
4.	1	0	X	00..010	00...00	mark trgt line arrived on first reference

Note that the case illustrated above is for a target page not resident in Ms. The same sequence holds for a ***near-miss*** (target *line* not in Ms) without marking all lines invalid at step 2. Step 3 illustrates the freezing of the buffer frame by virtue of not resetting the AL bit. The I/O buffer frame is effectively frozen by virtue of any non-zero bits in AL, i.e. any arriving lines. Only when the first reference is made to the target line is the frame unfrozen (unless other lines are arriving).

I/O buffer frames doing output are effectively frozen also by setting the IOF flag in the corresponding Ms map entry. If the MsC encounters a target MsDIR entry with IOF = 1, it continues scanning to the end of the hash chain, looking for an identical entry with IOF = 0. If such a copy is found, the corresponding frame holds the target. If no such copy exists the MsC interrupts the supervisor. The Ms page fault handler services this interrupt by invoking the replacement algorithm to select an Ms frame for the transaction causing the reference. The I/O buffer is then copied into the replacement frame. A delay of two disk accesses is thereby avoided by realizing that the referenced I/O buffer is still in shared memory. There is no consistency violation caused by two copies of the same object in Ms, if the transaction manager serializes commits, i.e. the two copies will be written back to secondary memory in the proper sequence. Note that at most *two* copies can exist because the MsC always searches the complete hash chain when mapping into an I/O buffer frame.

4.5.1 Allocation Policy

The allocation policy is based on the concept of a process ***pool***. Such a pool is a group of Ms page frames currently allocated to a process. The pool can grow and shrink according to the dynamic requirements of the process. Each pool has a dynamic maximum size limitation imposed by the resource allocation manager. The pool concept is useful because it structures the shared memory resources held by the active processes into fairly exclusive sets. Thus replacement is done in the pool belonging to the faulting process and the working sets do not trespass one another. This is not precisely true because sharing causes process dependencies not represented by the pools. LRU replacement over the whole shared memory, as is done in L2, has no regard for working set integrity. A process can grab as many pages as it wants using explicit I/O commands, possibly replacing large percentages of other working sets.

The pools are implemented by chaining together all the MsDIR entries corresponding to a single process. MsDIR.PL is the pool link pointer. Since there is only one link pointer per page frame entry, each data object resident in shared memory can be a member of only one pool. Therefore the number of page frames linked in a process pool is only an approximation to the real resources held by that process. It is predicted this method will not effect database applications adversely because the transactions will be short-lived, i.e. they will not exceed their maximum pool limitations. Scientific applications will be independent processes, so there will be little sharing. A special pool of page frames is kept, called the ***free pool***. Entries in this pool are always invalid and are the primary replacement candidates. They are created by aborted and committed transactions and transactions explicitly releasing temporary or shared pages.

The frequency of Ms faults, IFI_{mp}, is measured by a real time hardware device in the MsC. The resource manager examines IFI_{mp} periodically and if it falls below a certain threshold, the degree of multiprogramming is reduced. The number of active processes, usually called the ***multiprogrammed set,*** MPS, is reduced by adding the deactivated process pools to the free pool. The maximum pool size limitation of each active process is then increased [1].

Admittedly, this is a poor man's WS however, because the applications to be run on the system are highly variable, it is not known if the WS algorithm will work. WS is a local virtual time algorithm which dynamically manages the pool size of each transaction by discarding all page frames not referenced in the last time period (DENG80). Database applications with hundreds of short transactions and long scientific jobs have not been extensively modeled in the literature with WS replacement. WS and PFF (GUPT78), an affordable version of WS, are costly algorithms to implement, especially for a memory of 16K pages.

4.5.2 Replacement Algorithm

The basic replacement algorithm (fig. 4.19) first checks if the process causing the Ms fault has reached its maximum allowed pool size. If it has not, then a replacement page is taken from the free pool and added to the target pool. If the free pool is

[1] (CHAM73) analyzes allocation strategies such as this one, and presents an optimal initial allocation policy and near-optimal heuristic dynamic policies.

empty or if the target pool is full, then the algorithm searches the target pool for the best replacement choice. This search ends when a sufficient choice has been found, the search has exceeded its maximum time limit, or the complete pool has been searched. In the latter two cases, the current best replacement frame choice is taken.

The target pool search is conducted as follows. MsRs is accessed for each entry to find its access count, MsRs.AC. Page frames with MsDIR.BAD, DL, IOF or FF set are never chosen and frames with MsDIR.AL or MC bits set are given low replacement priority. All page frames not categorized in the above two groups, i.e. DL, AL,

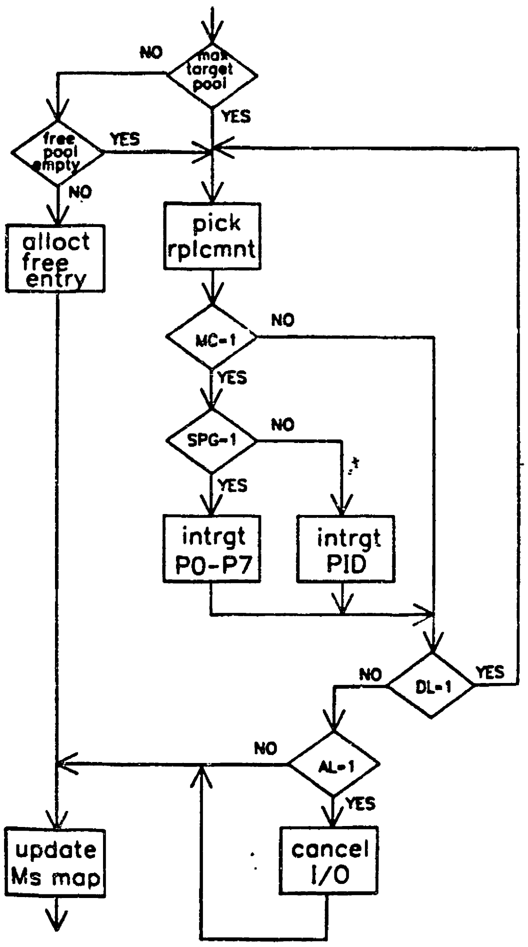


Figure 4.19. Ms Replacement Procedure

and MC = 00...00, are ordered by access count and the lowest access count chosen. If no candidate can be found in this manner, either a page with multiple copies or arriving I/O must be chosen.

If the former is picked the processor(s) must be interrogated for multiple copies. MsDIR.SPG indicates whether the page is part of a shared segment. This refers to segment type, not lock type. Non-shared pages are chosen over shared and R/O pages over R/W. The MC bit vector always indicates if multiple copies of each Ms line exist in the processor private memories. PID indicates the processor running the transaction which referenced the page if SPG = 0. The Ms page fault handler interrogates every processor if SPG = 1, or just processor PID if SPG = 0, for all 4KB children of the replacement VPA, using the INTL instruction. This causes all those data objects to be forced up to shared memory. During the interrogates for a R/W page, MsDIR.DL bits may be set indicating dirty lines. In this case the page cannot be replaced and another one must be chosen. For this reason R/W pages are rarely initially chosen for replacement.

If the page with an arriving line is chosen, the I/O request must be cancelled. It cannot be predicted which of the alternatives is worse.

If the degree of multiprogamming is 210 (30 transactions per processor), each pool is approximately 80 frames for equal allocation. The Ms map can be accessed in 50ns so that completely scanning the pool takes about 4us. This is sufficiently below the estimated I/O mean inter-arrival time of 100us (assuming IFI_{mp} = 10us and a 10% Ms miss ratio). This will not add to the private memory miss penalty if the MsC is given priority to access the Ms map store during mapping. Assuming only 15 transactions per processor and a 5% miss ratio, the replacement penalty is the same.

The last steps of the Ms replacement begin with the map update. The replaced frame lines are all marked invalid. The target line is marked arriving. If the target page frame is already resident in Ms, the only the target line is marked (arriving and invalid). Recall that an arriving line effectively freezes the page (section 4.5). The file manager is then invoked to perform the VA to BsA mapping (section 5.3) and the I/O manager is invoked to initiate the I/O.

5. File Management

This section describes the file manager, a member of the global operating system running on the supervisor which controls the virtual file system. Note that the terms ***file*** and ***segment*** are used synonymously. The instruction set supported by the file manager is not architected into the system. An earlier instance of the operating system support level, conceptually a virtual machine running on the supervisor, is the Ms page fault handler (section 4.5). The important file manager procedures are listed below.

ASSIGN__SID(filename)	: assigns an SID to new file
ERASE__SID(SID)	: frees up an SID
CREATE(SID,size,type)	: creates a file
LOOKUP(filename)	: returns the SID
OPEN(SID,window)	: activates the file window
CLOSE(SID)	: deactivates the file
APPEND(SID,size,type)	: appends to a file
BsMAP(VA)	: returns target BsA

These are supervisor state routines only, i.e. not directly executable by a user. The user interface is not discussed in this thesis. Essentially a user interface procedure takes the symbolic file name instead of SID as an argument and invokes LOOKUP and then the proper supervisor procedure.

There are four primary operations performed by the file manager:

- Assignment of a segment identifier, SID, to a symbolic file name on the creation of a file and the update of the free SID table on file erasure.
- Assignment of an I/O device address, BsA (physical address), to an SID on file creation.
- Mapping of a symbolic file pathname to its assigned SID on OPEN and CLOSE file instructions.
- Mapping of a virtual page address, VPA (page within segment address to be more precise), to its assigned BsA.

Secondary storage allocation is not discussed because it is highly dependent on the I/O device characteristics and applications particular to the system. The CREATE and APPEND procedures must efficiently allocate BsAs for all segment types in an effort to reduce disk contention. (JONE81a,b) analyze different allocation strategies such as *sequential*, *precessed* and *non-consecutive*.

Two major types of segments are distinguished. A segment is *sequential* if its records are (logically) referenced in sequential order and is *direct* if its records are referenced in random order. It is assumed that the file manager will allocate sequential disk sectors for logically sequential segments, although this is not a necessary constraint. File size is in increments of the virtual page size of 64KB, although disk allocation will be in smaller units of 4KB or less (multiple sectors). Each file sector within a virtual page is assumed to have a header containing forward and backward sector pointers, interpreted by the device controller hardware. In addition, each page in a sequential file has a header containing forward and backward page pointers (physical addresses), possibly crossing disk boundaries. Thus sequential files can be referenced in either the forward or backward direction and pages can be appended to either end. Within a page, references can be random; however, once a page boundary is crossed, the previous page cannot be reaccessed until after the file is CLOSEd and then reOPENed (section 7.3). Direct segments have the further classification of being *temporary* or *permanent*. Temporary segments are automatically erased when the transaction which created them has committed. Permanent segments can be explicitly erased with the ERASE__SID procedure. All segments are created with the CREATE procedure.

5.1 SID Assignment

The file manager keeps a ***segment descriptor table***, SDT, holding one entry for each of the 32K segments in the virtual space. The entries contain information concerning the segment, such as segment type [1]. Each entry is either linked into a circular free list or is in-use. The ASSIGN__SID and ERASE__SID procedures manage this aspect of the SDT. These procedures perform simple pointer manipulation functions.

5.2 Pathname to SID Conversion

Files are catalogued in a multi-level directory structure which is demand paged to memory like other segments. A specific design and implementation for the directory structure is not given here because alternatives are well documented (DORA76, SHAW74, HABE76). Basically, the structure is a tree consisting of directory file nodes and data (program) file leafs. The root node is the ***master file directory***, MFD. All other directory nodes are ***user file directories***, UFDs. A directory node contains the names of any offspring files. Thus a virtual file pathname is formed by concatenating the MFD name with the UFD names in the tree path leading to the data file leaf node corresponding to that file. A data file leaf node contains the SID of that file and pathname. Essentially, the file directory structure is used to translate a user pathname into a virtual file address. This permits users to deal exclusively with symbolic names for programs and data structures without worrying about machine usable addresses.

[1] Refer to Appendix C for SDT and other supervisor structure definitions.

5.3 VA to BsA Conversion

The method described here is similar to that of MULTICS but complications arise due to the possibility of very large segments and the large virtual page size. The mapping from VA to BsA depends on the segment type and size. On the average, small direct or sequential segments can be accessed fastest because mapping information is small enough to fit in a low level of the map hierarchy, as will be shown. The disk address [1] is assumed to be 8B long.

5.3.1 Segment Descriptor Table

At 32B per entry the SDT requires 1MB or 16 pages, which are frozen in shared memory to permit the mapping. The table is itself a segment, having an SID denoted as SID(SDT), and is addressed as follows

SID(SDT) | | SID(target) | | 00..00.

This virtual address points to the beginning of the target's SDT entry.

An SDT entry or segment descriptor consists of two parts, a ***header*** and a ***link*** (similar to the MULTICS branch pointer), each 16B. The header contains general information about the segment. The link contains one of a few things. If the file is sequential, the link contains the two physical addresses, BsAs, needed to access the file (the first and last page addresses, although only the first is needed if backwards accessing is disallowed). If the file is one or two pages long, the link contains the BsA(s). If the file is anything else, the link contains up to eight pointers to entries in the ***backup store address table***, BsAT. Each BsAT entry can hold a maximum of 1024

[1] Disk address, secondary memory address, backup store address (BsA) and physical address are used synonymously in this thesis.

BsAs. Thus the link indirectly contains the total number of physical addresses housing the segment.

The link pointers are dynamically allocated and deallocated, sequentially, by the file manager when needed for all non-sequential files of greater than two pages. The scheme is predicted to be efficient because most files will hold less than two pages (POWE77). In rare cases of large (greater than 128KB) files, the mapping requires an extra level of indirection. This indirection is not without penalty, however, because the BsAT is not frozen in Ms (it is too big). Therefore a disk access may result.

The link pointers are 2B each, which is not sufficient to index the BsAT, which can theoretically be spread over four segments. Therefore additional addressing information is needed, as will be described. The maximum BsAT size is calculated from the architected maximum of 2^{28} virtual pages. Each page requires an 8B physical address in BsAT, giving a maximum table size of 2GB! The BsAT will never, in a real system, get that big; however, the mapping facilities must be provided. Multi-segment files are not allowed in the proposed system, so that this maximum BsAT must be composed of four 512MB segments. Each BsAT segment can hold as many as 2^{16} entries. File manager policy is to always keep the physical addresses of a file in the *same* BsAT segment. That BsAT SID is kept in the SDT header as SDT.header.BSID. Therefore a link pointer requires only 2B.

An alternative partition is a 4MB SDT (64B per entry), allocating 48B per link. This larger link would either hold 6 BsAs or 12 pointers to the BsAT. The link pointer size increases to 3B (20 bits rounded to next byte) because the size of a BsAT entry is reduced to 64 BsAs = 512KB. This alternative organization reduces the size of the BsAT at the cost of quadrupling the size of the SDT. The mean time to OPEN a file is also reduced because now files up to 384KB in size can be directly addressed in the SDT. This design is suggested if it is felt that 4% of the shared memory can be spent on the SDT. Only the first organization is described below, although the alternative is similar.

5.3.2 Backup Store Address Table

The BsAT is directly indexed and has a fixed entry size of 8KB. An entry holds 1024 BsAs corresponding to 1024 consecutive virtual pages in the target segment. The effort so far was to minimize the SDT entry size because this table is directly addressable by SID and thus large and must be kept in Ms. This effort has been successful (SDT is only 16 pages) at the expense of a large BsAT. This is a result of having few pointers per SDT entry (8) and more physical addresses per BsAT entry (1024). The wasted space in the BsAT (a large percentage of BsAT is expected to be empty) does not directly impact system performance because the BsAT resides on disk. It is only paged into the memory when OPENing a file.

The ***indirect backup store address table,*** IBsAT, is introduced to aid in the explicit I/O needed when a BsAT reference misses. The IBsAT is a backup store address table for the BsAT (hence the name) and is frozen in Ms. The size of IBsAT is calculated as follows. The maximum BsAT size is 2GB or 32K pages. Thus the maximum IBsAT size is $32K \times 8B = 256KB$ or 4 pages. Again, normal system operation will never require more than one page for this structure.

A mechanism must be provided for reallocating BsAT entries and optimizing the distribution of these entries to keep BsAT size small. The ***backup storage address table usage map,*** BsATM, is a bit map of all entries in the BsAT. This map has a maximum size of four sections, each section holds 8K entries, each entry is 8 bits wide (32KB total). A section corresponds to a BsAT segment. An entry corresponds to a page in in the SDT. This design is suggested if it is felt that 4% of the shared memory can be spent on the SDT. Only the first organization is described below, although the alternative is similar.

5.3.2 Backup Store Address Table

The BsAT is directly indexed and has a fixed entry size of 8KB. An entry holds 1024 BsAs corresponding to 1024 consecutive virtual pages in the target segment. The effort so far was to minimize the SDT entry size because this table is directly addressable by SID and thus large and must be kept in Ms. This effort has been successful (SDT is only 16 pages) at the expense of a large BsAT. This is a result of having few pointers per SDT entry (8) and more physical addresses per BsAT entry (1024). The wasted space in the BsAT (a large percentage of BsAT is expected to be empty) does not directly impact system performance because the BsAT resides on disk. It is only paged into the memory when OPENing a file.

The ***indirect backup store address table,*** IBsAT, is introduced to aid in the explicit I/O needed when a BsAT reference misses. The IBsAT is a backup store address table for the BsAT (hence the name) and is frozen in Ms. The size of IBsAT is calculated as follows. The maximum BsAT size is 2GB or 32K pages. Thus the maximum IBsAT size is $32K \times 8B = 256KB$ or 4 pages. Again, normal system operation will never require more than one page for this structure.

A mechanism must be provided for reallocating BsAT entries and optimizing the distribution of these entries to keep BsAT size small. The ***backup storage address table usage map,*** BsATM, is a bit map of all entries in the BsAT. This map has a maximum size of four sections, each section holds 8K entries, each entry is 8 bits wide (32KB total). A section corresponds to a BsAT segment. An entry corresponds to a page in that segment and a set bit corresponds to an in-use BsAT entry. The BsAT manager must find up to eight zero bits in the BsATM when allocating BsAT entries to a new file. There is a trade-off between the size of the map (the density of ones) and the mean zero search time.

The file manager does not split segment allocation across BsATM sections. This results in all physical addresses for a given segment residing in one BsAT segment, and that is the segment indicated by SDT.header.BSID. The BsAT management algorithm attempts to keep the usage density (density of ones) constant in the BsATM. This implies there will be little variance in the length of a search during segment allocation. Searching for a zero continues after where the manager last found a zero. Deallocation on erasure of a file is performed by setting the bits addressed by the SDT link pointers to zero.

5.3.3 Active Segment Table

The ***active segment table,*** AST, is a table of physical addresses for mapping active segments. The AST permits fast access to recently used direct files just as the L2DLAT permits fast access to recently used private memory data objects. A direct segment is activated by OPENing it with the OPEN(SID,first__pg,last__pg) procedure. first__pg and last__pg are pages relative to the segment, indicating the active window. This is done because it is wasteful to totally activate a large file. For instance to completely activate a maximum size file of 8K pages, 64KB of the AST would be needed to hold the physical addresses. Since locality of reference is likely, only a locus or window of physical addresses will be needed over a moderate time period.

The AST speeds up the VA to BsA translation. Without the AST, a disk access would almost always be necessary to get the target BsA from BsAT. The AST, resident in Ms, holds this information and is accessed via the SDT. Therefore a VA to BsA translation for an active segment will require at most two shared memory accesses if the target BsA is in the active window.

Sequential files are activated with the OPEN(SID,FOB) procedure. The pre-page marker, SDT.header.SEQ.PPM, is set to either the first or last page of the file as indicated by the reference direction FOB (section 7.3).

The AST is restricted to few pages and must be managed efficiently. AST structure must allow rapid access of random BsA for direct access files. Since there are few large files expected in both scientific (POWE77) and database (RIES79) applications, it seems best to partition the management function away from the table and into the manager, i.e. keep the AST entries simple for space purposes thus requiring more complex addressing and reallocation algorithms. The AST is therefore organized as a variable length entry structure with an entry corresponding to an active segment window. Each active segment can have only one window, which can be dynamically reallocated, as access requirements grow and shrink. An entry contains a BsA for each consecutive page in the window. The size of the AST is thus a function of the mean number of desired active segments.

There are two instantiations of the OPEN procedure (fig. 5.1, 5.2): *explicit* and *implicit*. Sequential files must be explicitly OPENed so that the file manager knows the direction of reference. The user can explicitly OPEN a direct access segment window to set up the AST as mentioned but this is not required because a reference to a deactivated page of a segment will cause that page to be implicitly OPENed (transparent to the user). Implicit OPENs can reduce system performance because the file manager cannot "know" what the best window size is. A heuristic algorithm is

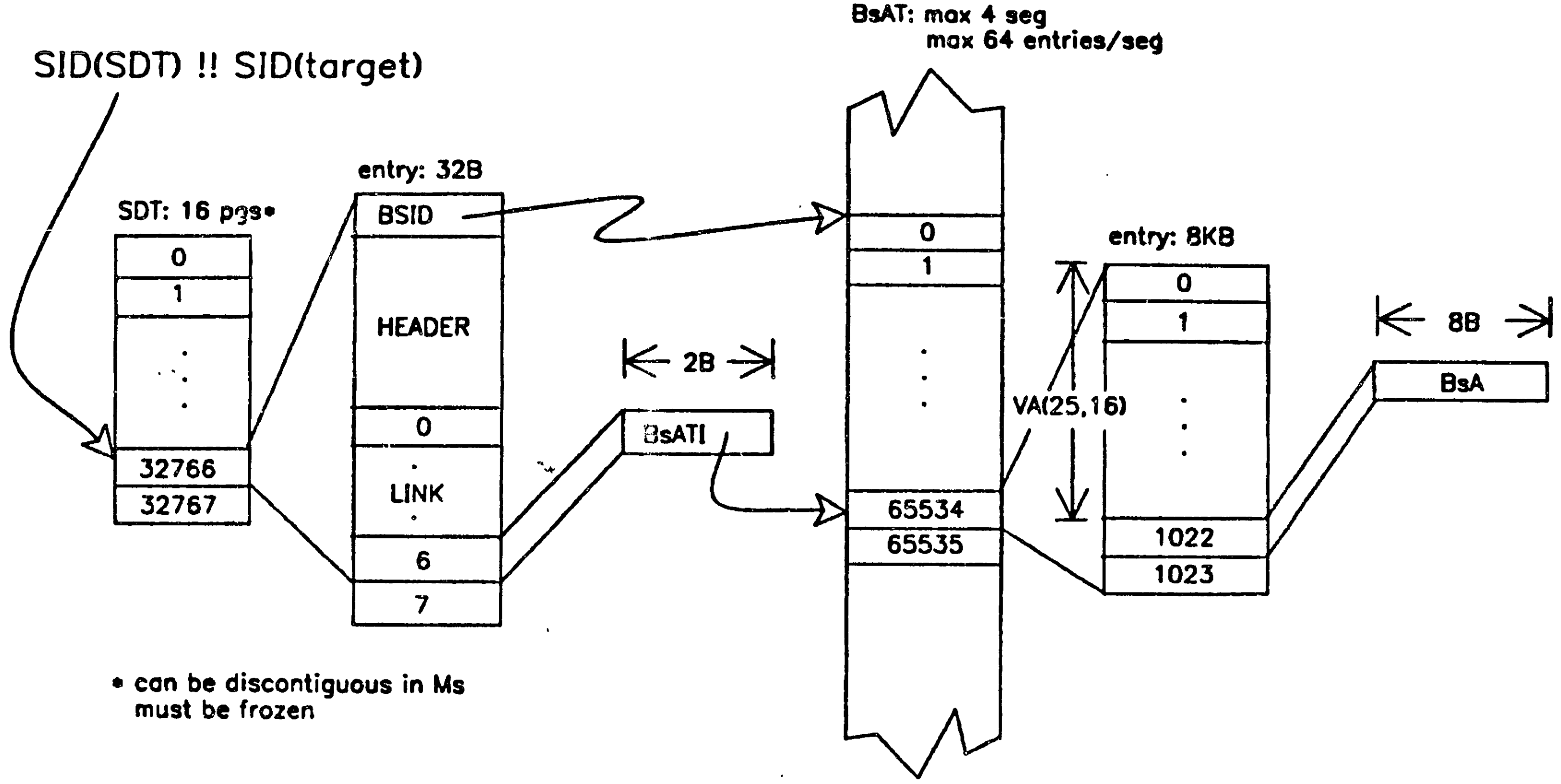


Figure 5.1. OPENing a Large Direct Access File

used for extending the window size implicitly. The new window is calculated as the union of the old window and

first__pg <-- target__pg - look__ahead

last__pg <-- target__pg + look__behind.

OPENing a direct file may cause another active file to be deactivated to free up enough room in the AST for the requested window. Replacement in the AST implies the possibility of thrashing within the AST. If the AST is observed to thrash, then the

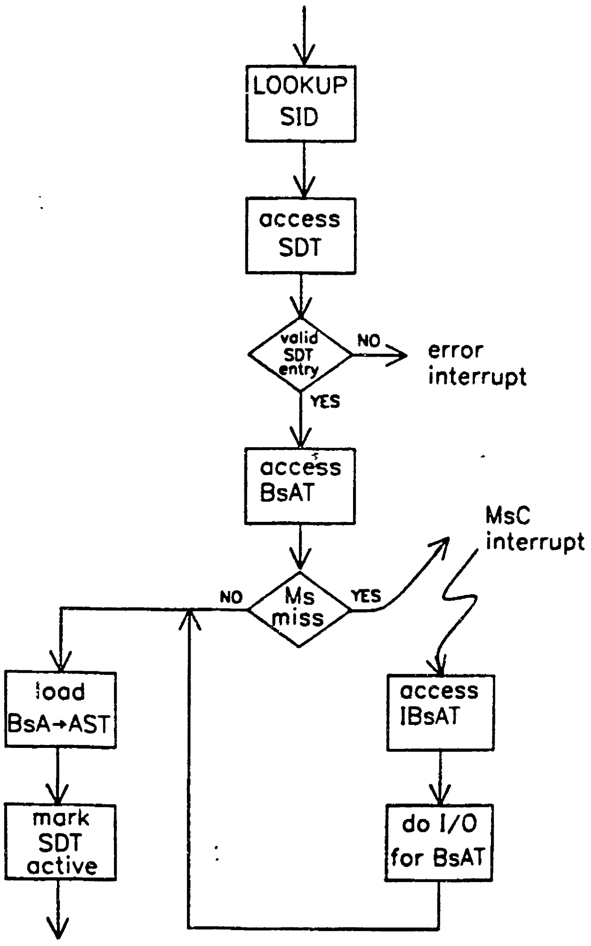


Figure 5.2. OPEN procedure

degree of multiprogramming is too high. The file manager must make decisions such as whether it is better to deactivate an old active file or reduce the window size of the requested file and how this policy should change for explicit and implicit OPENings.

The AST is frozen in Ms and its size is dependent on how much of the shared memory can be spared. Assuming it is eight pages, it will hold 64K BsAs. To address any AST entry requires 16 bits. Thus the size of a hole between AST entries also requires 16 bits. These holes are created by the deactivation of segments. The holes are circularly linked (HABE76) and a hole descriptor requires 4B for size and link pointer fields. The management algorithm is FFC, first fit chosen (HABE76).

A high level description of the VA to BsA map for a direct access file follows (fig. 5.3, 5.4). The Ms page fault handler accesses the SDT target entry. If the entry is invalid, the file manager returns that information. If the entry is valid, the segment exists and the header information indicates if the file is active in the target range, i.e. that range has been OPENed. If so, the AST is accessed by SDT.header.ASTI and indexed by the VA. The target AST entry holds the BsA. If the file range is not active then the handler invokes an implicit OPEN with the new range. This will update the target SDT entry header first and then access the BsAT with

SDT(target).header.BSID | | SDT(target).link(VA(28,26)).

The referenced page of the BsAT may not be resident in Ms, in which case the MsC issues an interrupt to the supervisor. The Ms page fault handler realizes the VA that missed was part of the BsAT segment and thus indexes the IBsAT for the target BsAT entry disk address (recall the IBsAT is frozen in shared memory). The I/O manager is then invoked to fetch the BsAT entry. Once the BsAT entry is accessible, the appro-

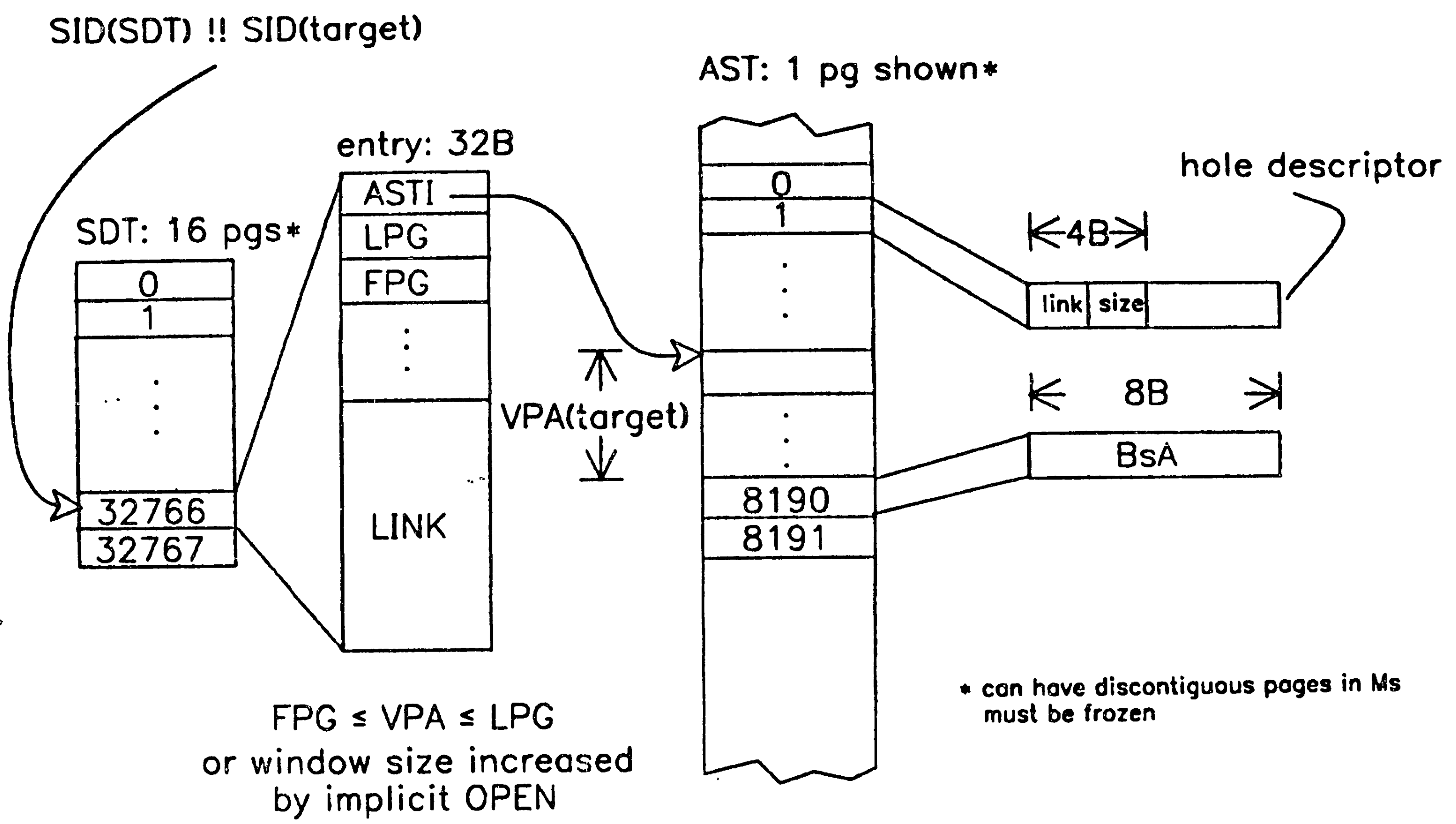


Figure 5.3. VA → BsA Mapping

priate number BsA entries are written into the AST. This may require AST compaction and/or replacement. The target BsA is returned.

The CLOSE procedure marks the SDT entry inactive and adds the associated AST entries to the free list as a hole. A sequential file must be CLOSEd before pages previously referenced can be referenced again.

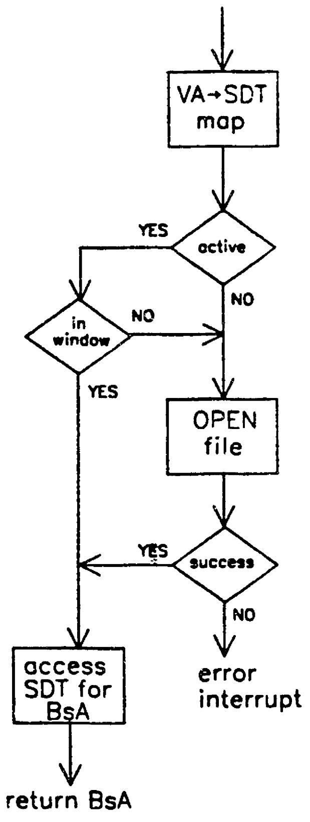


Figure 5.4. VA → BsA Mapping

6. I/O Management

The I/O manager is responsible for initiating both *implicit* and *explicit* I/O requests to secondary storage. Implicit input is equivalent to demand paging. Explicit input is usually for larger data objects. Explicit output is disallowed in the BIP MP database environment by consistency issues. Implicit output is equivalent to committing. For applications that do not require consistency, explicit output is allowed. Implicit output is essentially shared memory write-back and is invoked by the Ms fault handler. The output and input protocols are similar. Input is briefly described.

6.1 Input

6.1.1 Demand Paging

The I/O manager takes the following steps to service an input request from the Ms page fault handler. The Ms page fault handler passes it the BsA of the target data object and the MsA of the replacement frame. The target virtual 4KB line address is used to determine the proper track and sectors of the disk. Note that a 4KB disk block transfer size on demand paging is necessary in the database environment because the smallest lockable unit is 4KB (section 8.1). Therefore if a larger data object were implicitly delivered, it might overwrite modifications made by another lock holder. A larger demand paged object is permitted if consistency is not guaranteed.

The I/O manager then writes the channel program with the target BsA and I/O buffer frame MsA. This program is either written into Ms or sent directly to an intelligent channel device. The I/O is then initiated. At some point the channel issues a disk request, routes the transfer to the Ms frame and at completion interrupts the I/O manager.

The I/O manager marks the I/O buffer frame target line as valid and invokes the scheduler to requeue the I/O bound transaction in its corresponding processor queue. When the transaction finally gets switched in, the reference that originally caused the I/O will be re-tried. The reference will hit because the line is valid and the MsC will mark the line as arrived, effectively unfreezing the frame.

6.1.2 Explicit Input

Explicit input permits several lines to be requested from secondary memory at once. The I/O manager must first check that the requested pages are not already present in the shared memory by scanning the proper transaction pool in the Ms map. Explicit input makes the Ms replacement algorithm more complex because multiple frames must be allocated in one invocation for efficiency. The resource allocation manager is invoked to determine if the I/O request has the proper authority to expand the pool size of requesting transaction, at the expense of possibly deactivating other transactions. The I/O manager then invokes the replacement algorithm to select the desired number of page frames from among either the target pool or free pool. The I/O is then initiated as described above.

6.2 Burst Multiplexor Channel Device

The channel device goals (section 1.1) were accomplished through a series of trial designs. The final design is described by outlining its evolution. Note that a complete channel device is not detailed; only the dataflow is given. The issues of controller design needed to make the channel device "intelligent" are not dealt with either.

An ***I/O channel device*** is a hardware unit which converts several ***channels,*** low bandwidth data streams from ***I/O devices,*** into a single high bandwidth stream to memory. For simplicity, each I/O device is assumed to be integrated with a dedicated ***control unit.*** Only serial output movable arm disks are considered. Note that this terminology is different from (IBM81). Essentially an IBM "channel" is called a channel device here and an IBM "control unit path to a subchannel" is called a channel or referred to as the associated disk arm.

6.2.1 Forward Path

In essence, the forward path is a serial-to-parallel converter which time multiplexes several I/O devices to increase throughput. An internal store is needed to produce the high output bandwidth by buffering the data and *bursting* complete data blocks onto the bus. This serial-to-parallel converter is called the ***speed matching buffer.*** The larger internal store is called the ***burst buffer.*** Together with the channel selecting switch these are the main components of a ***burst multiplexor*** or ***switching buffer channel device.***

A channel device is specified as C(n,m). The number of attachable channels, n, is called the ***channel set.*** This is the maximum number of serviceable channels. The number of simultaneously active channels, m, is called the ***active channel set*** (IBM subchannels), and is also the number of simultaneous active disk arms. This is the maximum number of attached channels that can be active at any one time ($m \leq n$). For m < n, a switch is needed somewhere in the device to select among the attached controller design needed to make the channel device "intelligent" are not dealt with either.

An ***I/O channel device*** is a hardware unit which converts several ***channels,*** low bandwidth data streams from ***I/O devices,*** into a single high bandwidth stream to memory. For simplicity, each I/O device is assumed to be integrated with a dedicated ***control unit.*** Only serial output movable arm disks are considered. Note that this terminology is different from (IBM81). Essentially an IBM "channel" is called a channel device here and an IBM "control unit path to a subchannel" is called a channel or referred to as the associated disk arm.

6.2.1 Forward Path

In essence, the forward path is a serial-to-parallel converter which time multiplexes several I/O devices to increase throughput. An internal store is needed to produce the high output bandwidth by buffering the data and *bursting* complete data blocks onto the bus. This serial-to-parallel converter is called the ***speed matching buffer.*** The larger internal store is called the ***burst buffer.*** Together with the channel selecting switch these are the main components of a ***burst multiplexor*** or ***switching buffer channel device.***

A channel device is specified as C(n,m). The number of attachable channels, n, is called the ***channel set.*** This is the maximum number of serviceable channels. The number of simultaneously active channels, m, is called the ***active channel set*** (IBM subchannels), and is also the number of simultaneous active disk arms. This is the maximum number of attached channels that can be active at any one time ($m \leq n$). For m < n, a switch is needed somewhere in the device to select among the attached disk arms for the set to be serviced. The term switch is used loosely here, combining the functions of a multiplexor and router.

The initial design was formulated as a minimal effort, brute-force configuration (fig. 6.1). The switch was left out for simplicity. Rough calculations (section 1.2.3) showed 32 to 150 simultaneously active disk arms for those specific applications. A database application will most likely have a lower number of simultaneous arms and a larger number of total arms. In both cases, in consideration of upgrading the I/O system in balanced increments, a C(64,64) channel device was chosen.

6.2.1.1 Speed Matching Buffer

The speed matching buffer, SMB, is a set of shift registers, two per active disk arm (fig. 6.2). These registers are called SMB lines, A and B for each active disk arm. The shift registers are built out of shift chips, designed in the assumed technology (fig. 6.3). The chip partition is a 16 bit serial input, 8 bit parallel output shift register set and is I/O limited.

Each disk arm supplies the SMB three input lines: datum and two phase clock. Clock lines for different channels are asynchronous permitting each to operate independently. The shift registers are clocked by the associated disk clock lines, advancing the data through the register with a minimum 20ns full cycle (48 MbS). The first pass design specified 32B long registers, requiring $64 \times 2 \times 32 = 4K$ shift chips. Each clock set increments a private line counter, which indicates the progress of the input. When the counter overflows, the channel controller switches the input stream to the alternate line. The controller scans the private line counters in order and off-loads any full line, 32B in parallel, to the burst buffer.

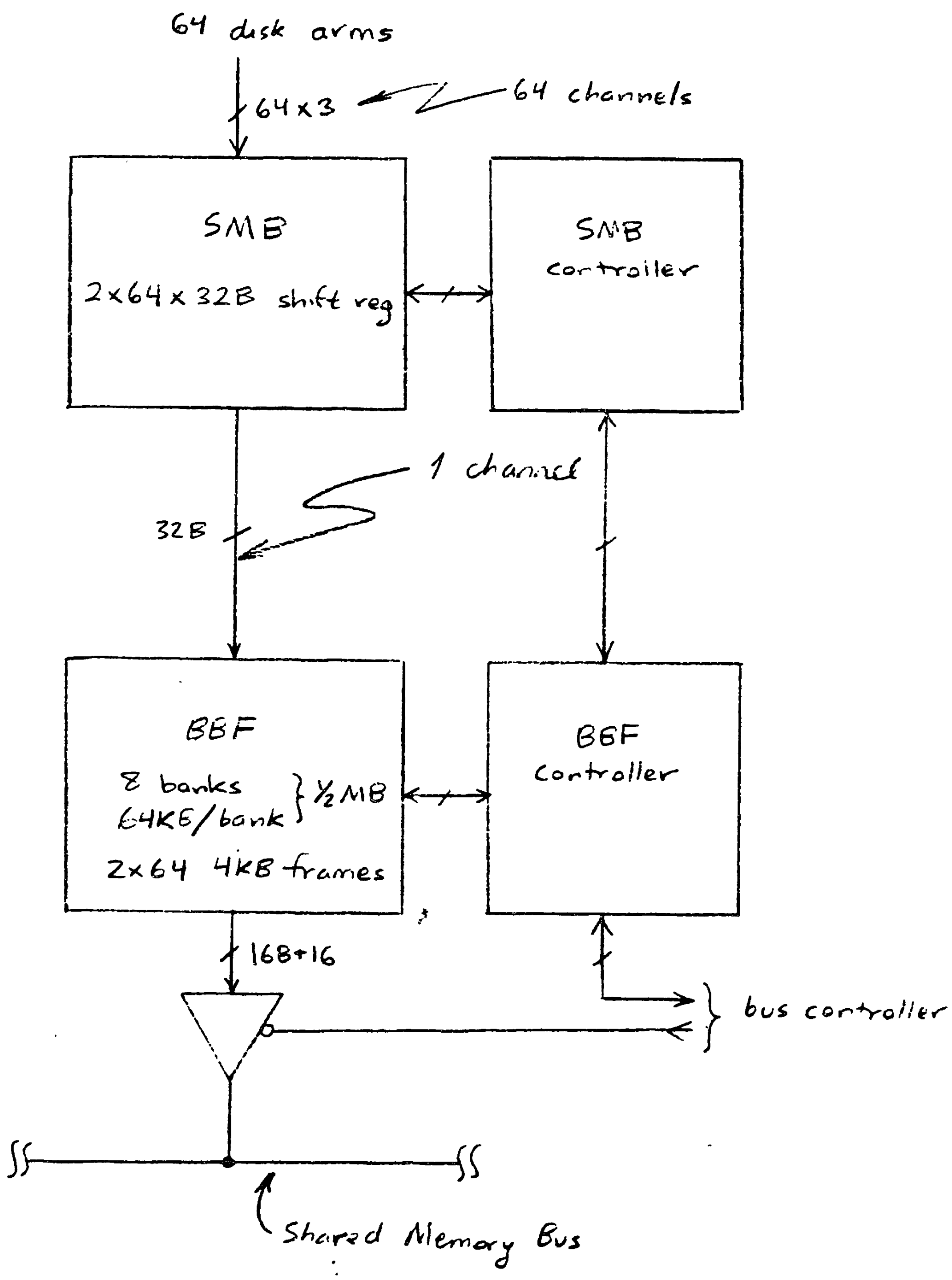


FIGURE 6.1. BMC: First pass design

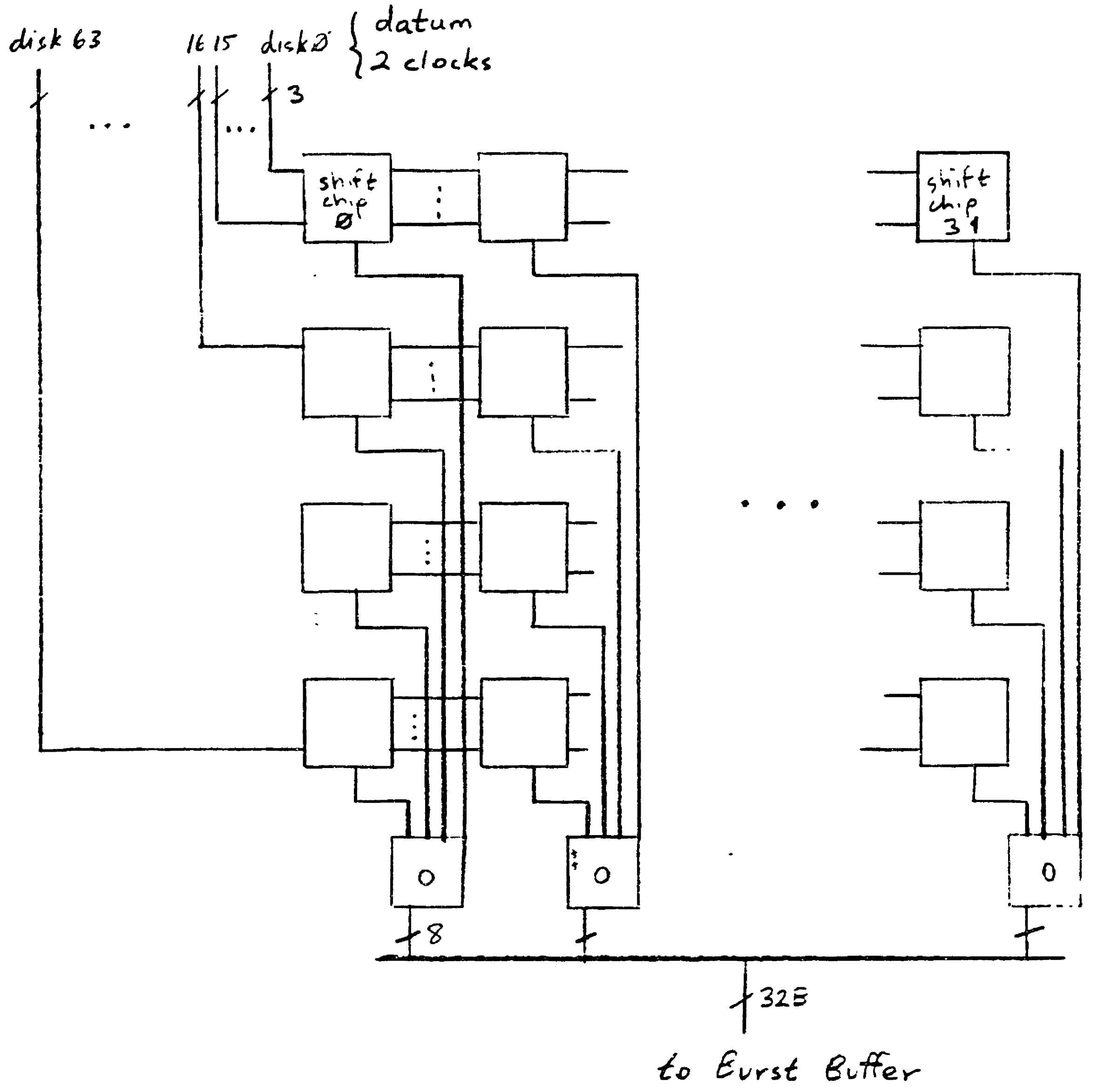


FIGURE 6.2. SMB dataflow: brute force design

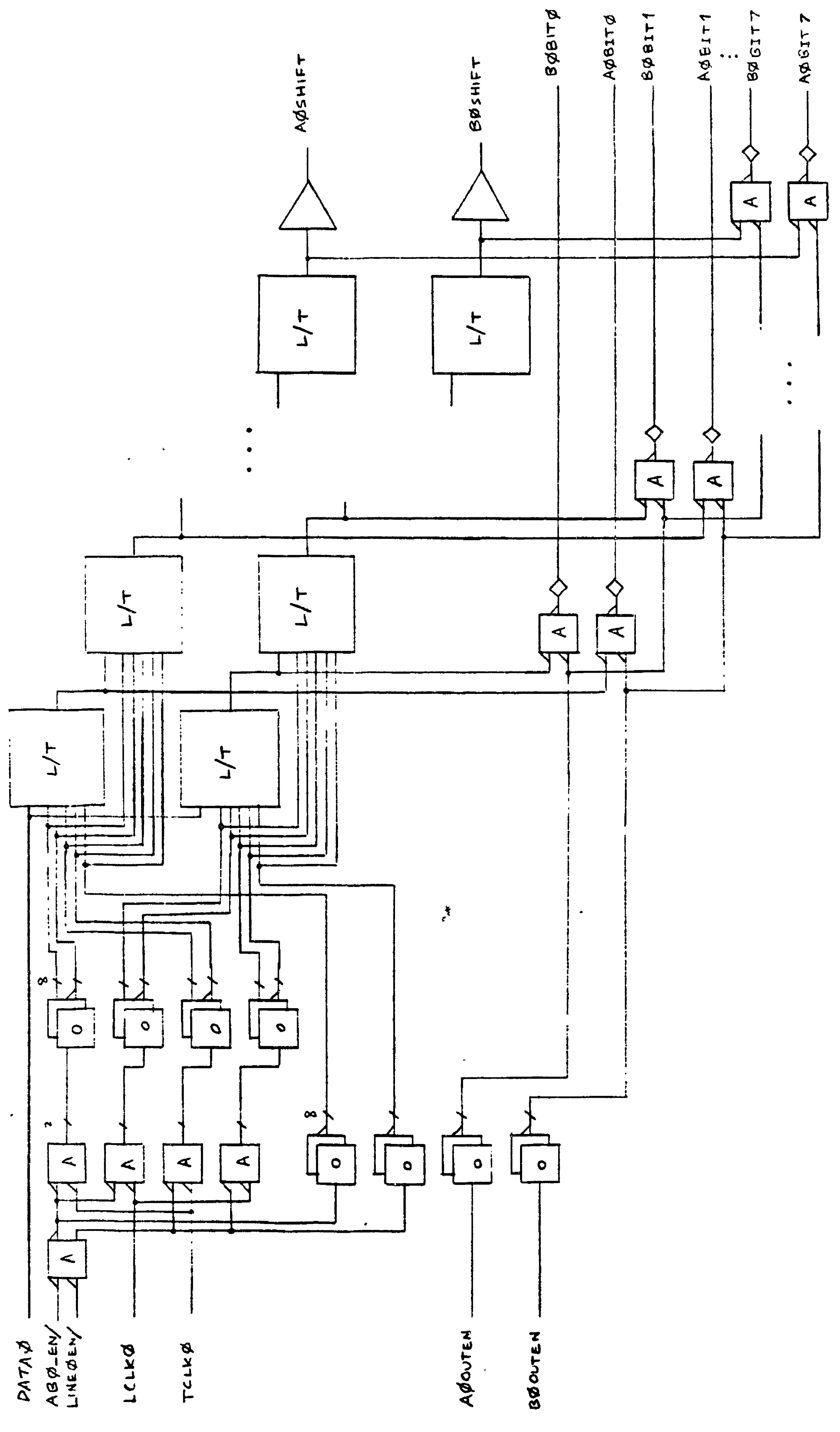


FIGURE 6.3a. Shift Chip : 1 of 16 Dual Byte Lines

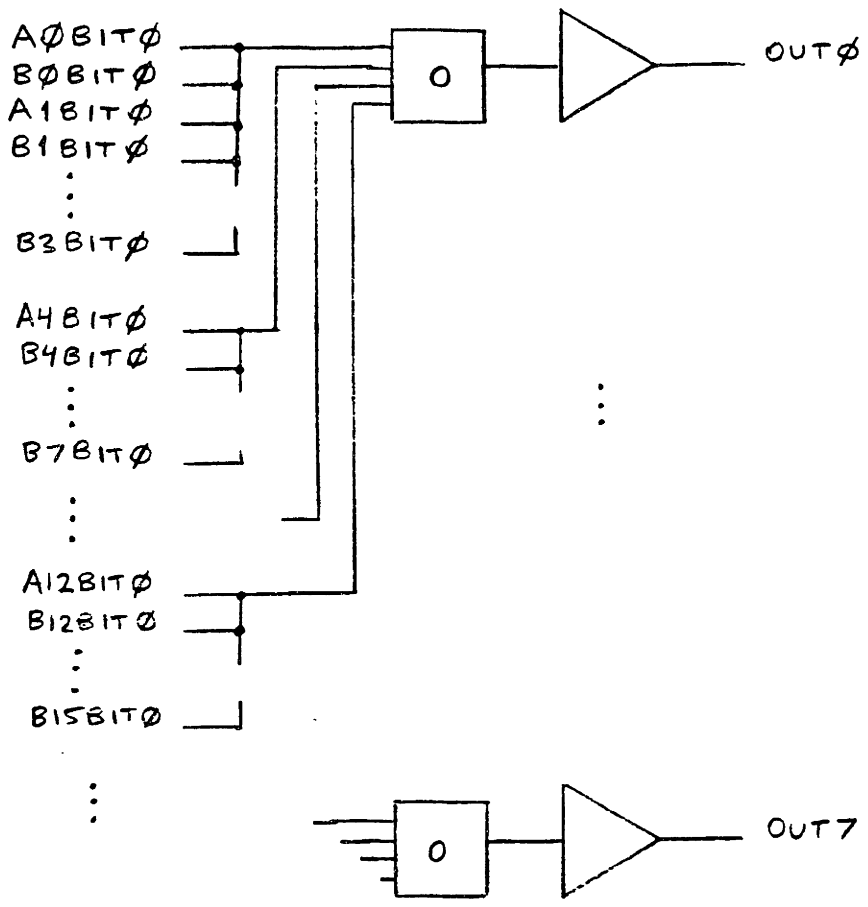


FIGURE 6.3b. SHIFT CHIP : PARALLEL OUTPUT

The timing constraint for this off-load operation, as seen from the SMB vantage point, is that the controller must off-load 64 lines before the dual 64 fill up. This assumes that the disk arms are acting synchronously (worst case behavior) and implies (32B / 6MBS) / 64 = 80ns per off-load. This is no problem for the SMB, which has negligible write delay. It will be shown, however, that the burst buffer cannot support this rate using the assumed technology.

6.2.1.2 Burst Buffer

The burst buffer, BBF, is a temporary store for collection of incomplete transfer blocks on their way to the shared memory. There is a trade-off between I/O request throughput and shared memory utilization, both of which are desired to be large. R_{eff}, the effective I/O rate (section 1.2.1), decreases as transfer size decreases as shown below.

b (KB)	**R_{eff} (KBS)**
1	51
2	101
4	200
8	380
16	720
32	1300
64	2200

Thus decreasing transfer size to increase I/O request throughput will decrease the I/O utilization of shared memory. The transfer size chosen for this design is 4KB. Larger block accesses using one seek can be accomplished by a more complex channel controller - there is no inherent limitation in the dataflow. The burst buffer 4KB frame size does not affect the explicit I/O capability.

This buffer is 4-way set associative. Similar to the SMB, there will be 2m frames, where m is the number of active disk arms. A worst case senario indicates that m frames must be offloaded to the shared memory before the other m frames fill up. If this is not accomplished, the input streams must be inhibited, reducing throughput. In general, it is difficult to accurately predict the two factors influencing channel throughput: frequency distribution of disk utilizations (source) and the service time distribution of the shared memory (sink). Therefore sizing the BBF at 2m frames is hard to justify.

If enhancing I/O throughput were the only design consideration, the BBF could be mapped by sector, i.e. an active disk arm maps directly to two unique BBF frames. But consider the case when only a few disk arms are active. In an effort to reduce shared memory contention, a set associative BBF will permit the few disk arms to be serviced in a less critical manner. If the channel device is serviced before the active channels fill up their respective classes, no loss of throughput occurs. A fully associative BBF is of course optimal performance-wise, but the control is too expensive. It is also hard to justify the set associativity of the BBF without extensive simulations.

The first pass design calls for a BBF of $2m = 2 \times 64 = 128$ $4KB$ frames = 512 KB. Using double density L2 technology, the BBF requires $(512KB \ / \ 2K) \times 1.125 = 288$ chips. The additional one chip per byte is for parity.

The BBF is interleaved to produce a 16B output line for the shared memory every 16ns. Assuming a 16ns cycle channel clock, the BBF must sustain a rate of one output line per cycle, when bursting output. During the burst write the BBF controller cannot neglect channel input. A read and a write are therefore done on alternating four cycle slots (fig. 6.4). 4 cycles x 16ns/cycle = 64ns a conservative estimate of BBF access time (50 ns + 30%). The reads are hidden under the writes by the 8-way interleaving. Beneath each read-write access, eight 16B line transfers are issued from the staging registers. One complete 4KB page is burst in 256 cycles or 64 read-writes.

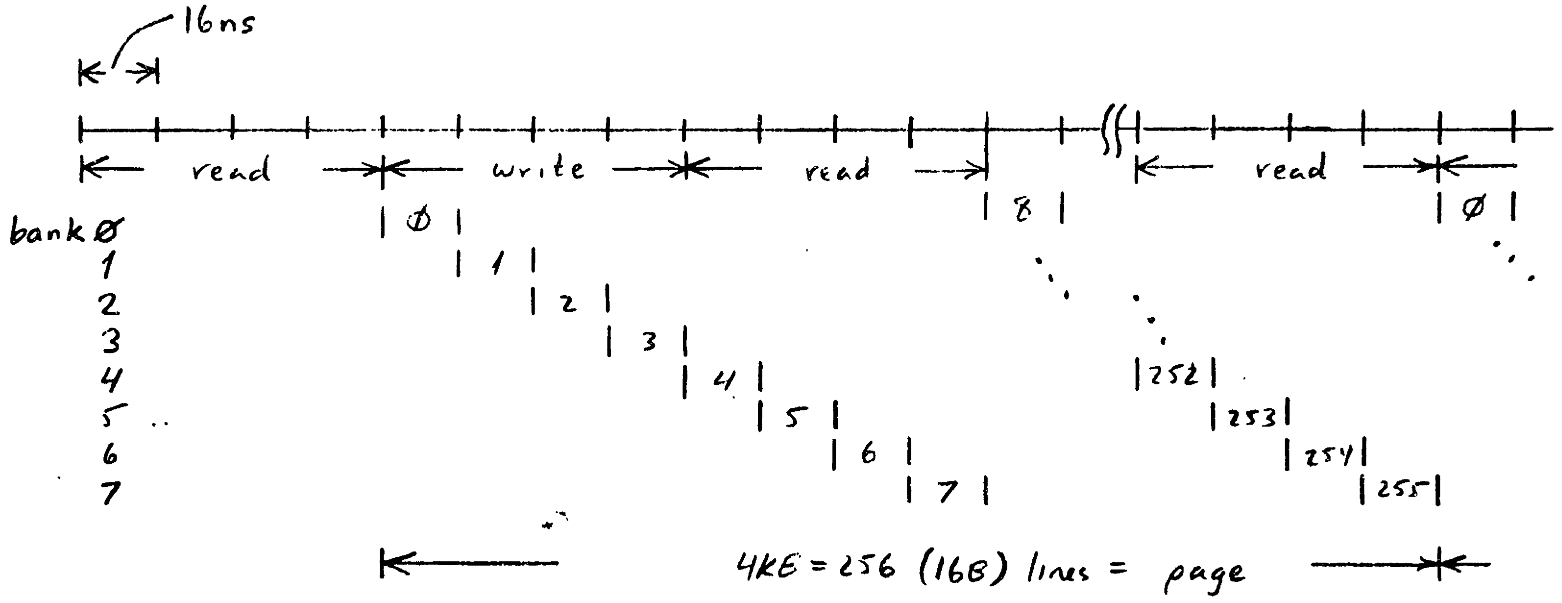


FIGURE 6.4. BBF timing: 1GBS output rate

Previous calculations show however that the combined sum of the 64 active channels will produce a full SMB line every 80ns. The BBF can support only one read every 130ns, as previously detailed. Therefore the 64 channels are supplying too much bandwidth for the BBF. The channel device design described can sustain only 64 × (80/130) = 40 active channels. This problem must be solved by redesigning the dataflow. A solution involving more complex control, such as deactivating a channel if it cannot be serviced by the BBF, is rejected because it will result in lower throughput per channel. This is expected because the transmitting channel which is suddenly blocked will have to re-send the complete 4KB data object.

The first fix is to increase the SMB line length to 64B. This will lower the BBF input rate by half, to one read every 160ns. The number of SMB chips increases by a factor of two, to 8K. The previous BBF design, with its eight banks, each 16B across, can handle this new input width.

The second fix is to introduce a switch to decrease the number of active disk arms. Theoretically, a maximum of 40 channels can be switched in and still satisfy the present BBF timing constraints. This is not practical because the switches with power of two inherent blocking are much simpler to build and control. A switch is specified by a triplet, S(i,j,k) where there are i input sets and j output sets ($i \leq j$). In this thesis, the ***inherent blocking*** is defined as the ratio i / j. The ***blocking ratio,*** k, is defined as an indicator of the "flexibility" of the switch. A low blocking ratio means most inputs can be routed to most outputs. A high blocking ratio implies a rigid, yet simple switch. The special case of k = 1 is called a ***full switch.*** This is expensive in hardware, but has the advantages of simple control and complete flexibility. A major design criteria was to partition the hardware in such a way as to avoid large switches. Small switches are desirable for two reasons. First, the cost penalty of using a small

Previous calculations show however that the combined sum of the 64 active channels will produce a full SMB line every 80ns. The BBF can support only one read every 130ns, as previously detailed. Therefore the 64 channels are supplying too much bandwidth for the BBF. The channel device design described can sustain only $64 \times (80/130) = 40$ active channels. This problem must be solved by redesigning the dataflow. A solution involving more complex control, such as deactivating a channel if it cannot be serviced by the BBF, is rejected because it will result in lower throughput per channel. This is expected because the transmitting channel which is suddenly blocked will have to re-send the complete 4KB data object.

The first fix is to increase the SMB line length to 64B. This will lower the BBF input rate by half, to one read every 160ns. The number of SMB chips increases by a factor of two, to 8K. The previous BBF design, with its eight banks, each 16B across, can handle this new input width.

The second fix is to introduce a switch to decrease the number of active disk arms. Theoretically, a maximum of 40 channels can be switched in and still satisfy the present BBF timing constraints. This is not practical because the switches with power of two inherent blocking are much simpler to build and control. A switch is specified by a triplet, S(i,j,k) where there are i input sets and j output sets ($i \leq j$). In this thesis, the ***inherent blocking*** is defined as the ratio i / j. The ***blocking ratio,*** k, is defined as an indicator of the "flexibility" of the switch. A low blocking ratio means most inputs can be routed to most outputs. A high blocking ratio implies a rigid, yet simple switch. The special case of k = 1 is called a ***full switch.*** This is expensive in hardware, but has the advantages of simple control and complete flexibility. A major design criteria was to partition the hardware in such a way as to avoid large switches. Small switches are desirable for two reasons. First, the cost penalty of using a small full switch is affordable. Second, small switches are easily partitionable in the assumed technology. A S(64,32,1) switch is a good candidate, implying a C(64,32) channel. The switch design is partitioned into 32 to 1 three bit MUXs. The MUX design is purely combinational logic and fits on a chip. It is I/O limited at 32 $\times$ 3 inputs and 3 outputs. Two of these chips comprise a 64 to 1 MUX so that the switch requires 32 chips.

The second solution is chosen for the following reasons. Primarily, the switch reduces the size of the channel roughly by a factor of its inherent blocking because less channels have to be served concurrently. For C(64,32), the BBF is only 144 chips. For C(64,16), not only is the BBF 72 chips, but the timing constraints are sufficiently reduced to permit a 16B SMB line, meaning a SMB of 512 chips.

An additional advantage of the second solution is that it supports ***fault tolerance*** in a simple yet elegant way. Since the cost of the channel is approximately proportional to the number of active disk arms in this range, redundant lower capability channel devices can be used with the same cost performance as a single high capability channel device. A ***gracefully degrading system*** is created at no significant extra cost by allocating the same set of channels to a group of smaller channel devices. Consider four C(64,16) channel devices, each front-ended with a S(64,16,k) switch. The cost of these device, less the switches and minor additional control, is equal to the cost of the C(64,64) device. Dividing a 64 channels among them, each will service its own subset of 16. When one fails, the others take up its work load. Reliability has been increased at the cost of performance.

The final design calls for two C(64,16) channel devices, each with a S(64,16,1) switch, per 64 disk group (fig. 6.5). The switching gives an effective C(64,32) device

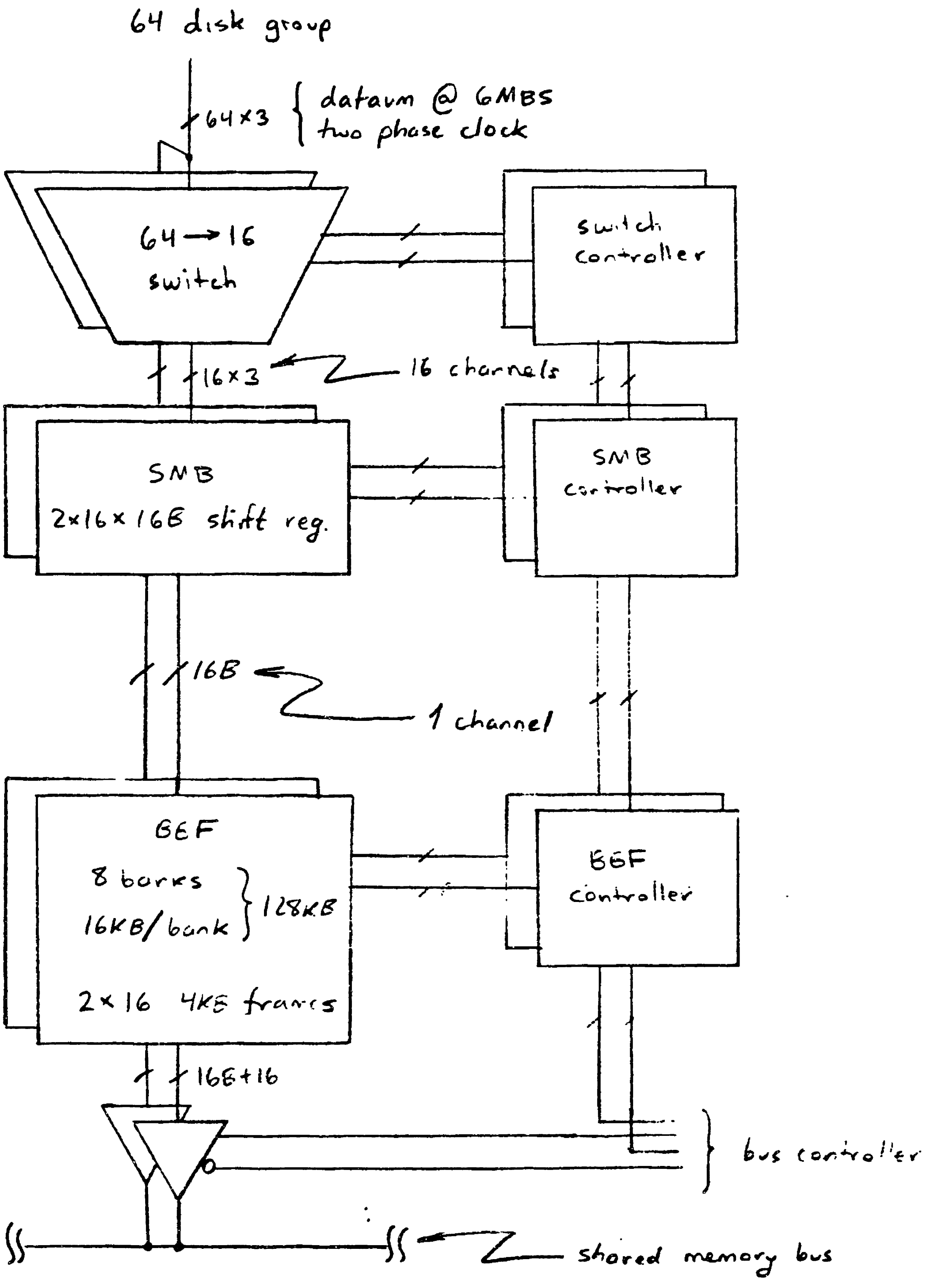


FIGURE 6.5. BMC: final design

capability during normal operation. In degraded mode, it assumes a C(64,16) capability. The final design data rates are illustrated in fig. 6.6.

6.2.2 Reverse Path

The reverse path dataflow is similar to the forward path, except for two major alterations.

- The switch is converted into a router, i.e. S(16,64).
- The shift chip holds 16 parallel load 8 bit shift registers with serial input and output shift lines.

The reverse operation is as follows. First the BBF fills up a frame from the bus on a supervisor initiated transfer from shared memory. When the frame is full, it is loaded, 16B at a time, into the selected channel's SMB line. After an A line is loaded, the BBF next loads the corresponding B line. The A line is concurrently shifted out in serial to the disk arm at 6MBS. The switch acts as a router connecting each of the 16 SMB lines to any of the 32 channels.

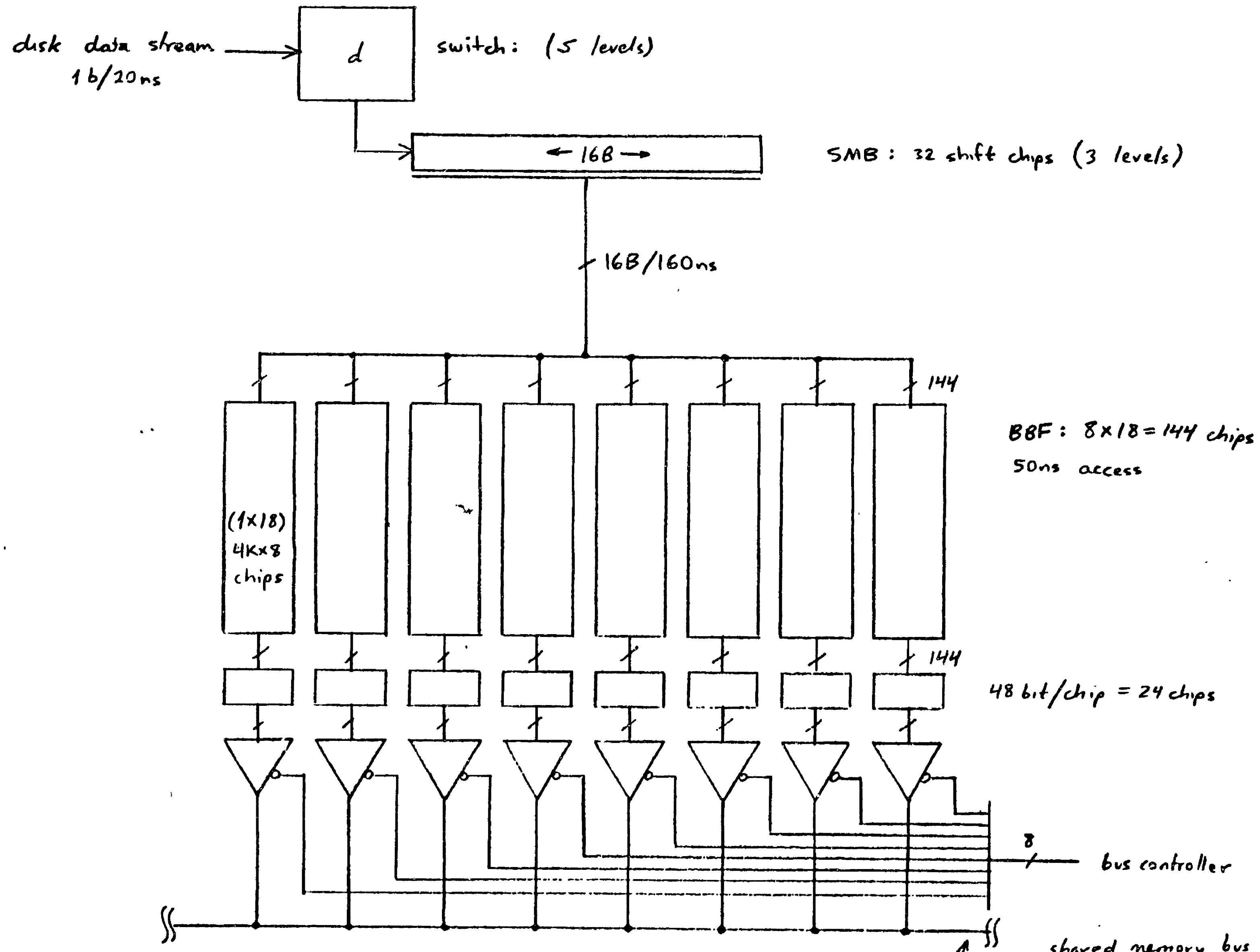


FIGURE 6.6. Final Design Data Rates

7. Additional Points

7.1 Shared Memory Design

The shared memory was designed around the following goals:

- satisfy performance requirements, namely support a 1GBS bus rate.
- low miss ratio.
- fault-tolerant design - partition Ms into as many independently faulting modules as possible. This involves partitioning the logical memory so that a reasonable number of hard errors (chip failures) cannot render the complete Ms address space inoperable.

The three goals are equally important. Unfortunately, they are all costly. The cornerstone of the memory design depends on the bus width. This determines cycle time, the number of bus cycles needed for a block transfer and therefore the degree of memory interleaving necessary to support a 1GBS bus rate. Assuming a 16B wide bus, a 16ns cycle is needed. A 1KB transfer requires 64 bus cycles or 1us. This is the assumed Ms packaged access time, therefore Ms must be 64-way interleaved.

The width of each Ms bank will depend on the available form factors of the dynamic RAM chips and the desired shared memory size. Ms size will in turn effect hit ratio. The following table displays the relationship between possible form factors and the resulting number of independently faulting memory modules for a 1GB Ms.

form factor	# modules	#chips
256K x 1	4	512
128K x 2	8	512
128K x 1	8	1K
64K x 4	16	512
64K x 2	16	1K
64K x 1	16	2K
32K x 8	32	512
32K x 4	32	1K
32K x 2	32	2K
32K x 1	32	4K

Note that the vertical form factor, vff, is inversely proportional to the number of modules:

$$modules = \frac{2^{20}}{vff}.$$

Each module is essentially a row of chips in the memory array. It is assumed an additional error detection and correction bit store is kept in parallel with the Ms store to permit single error correction and double error detection. Repeated double error detections at certain related locations in the Ms address space imply a single chip had a hard failure. The detector will interrupt the supervisor which will execute a low-level recovery procedure (section 10). From that point on, the system ignores that module, so that it can be physically replaced. The shared memory organization is illustrated in figure 7.1.

The shared memory can theoretically have multiple ports, one per module. The memory units could then be configured with the processors and channel devices in a large switch. This type of configuration implies that novel approaches to replacement are needed to reduce memory interference (SMIT77). These issues are not discussed because the switch is extremely expensive and not believed to have a significant cost/performance advantage. For instance a crossbar (full) switch must connect the eight processors to a minimum of four memory modules and say four channel devices.

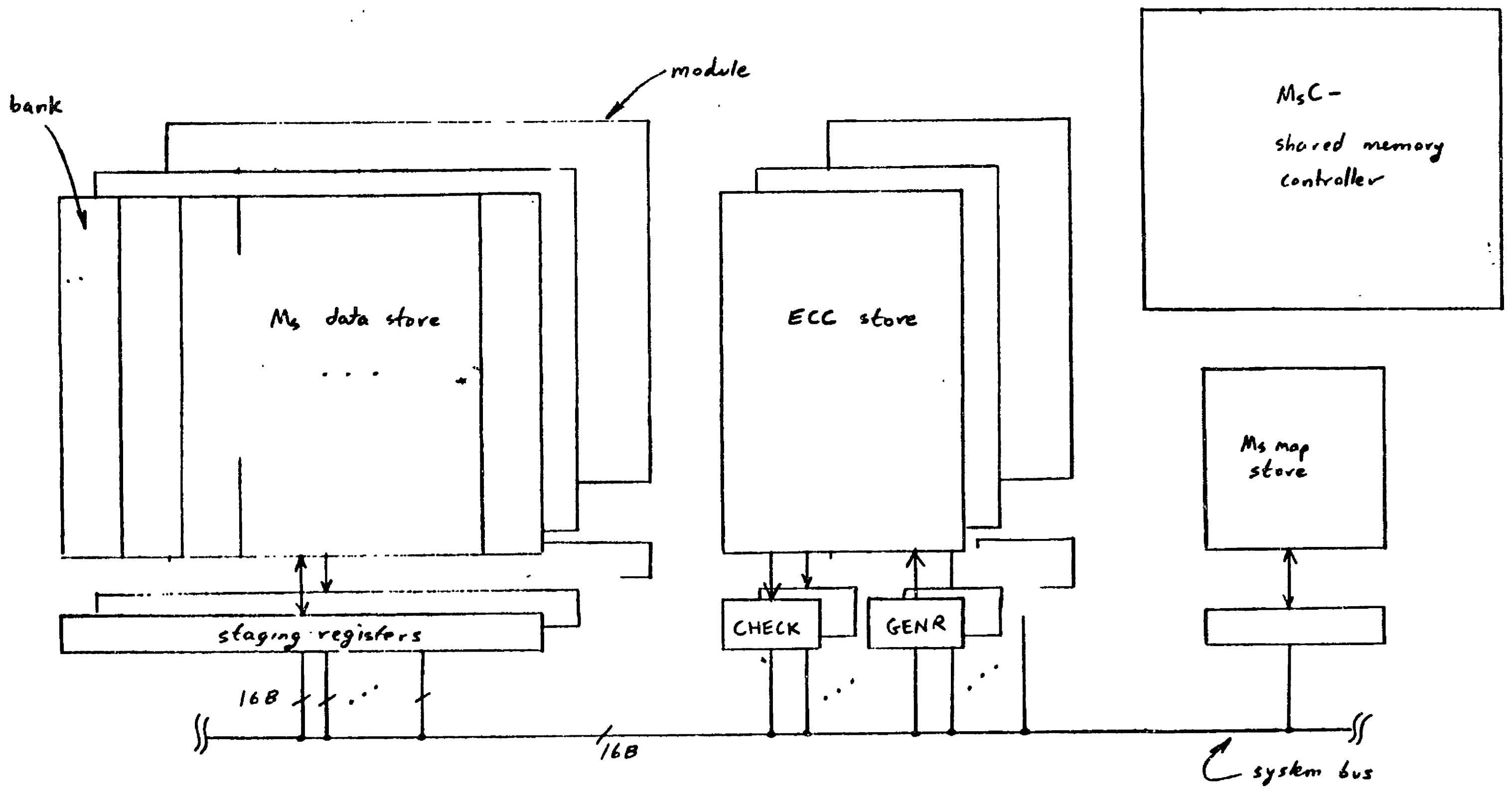


FIGURE 7.1. Shared Memory Overview

This involves 64 cross points for 16B wide paths! If eight modules are used, 96 cross points are needed! Construction of one 1GBS bus is hard enough without attempting to add this many mechanical contacts and additional delays.

7.2 System Bus Design

As concluded from the queueing model, the bus is simply an extension of the single port shared memory. The purposed bus width is 16B. Whether it is reasonable to design a 16ns cycle bus can be decided only with better defined packaging constraints. Long buses can be sped up a number of ways, for example by periodic latching, i.e. pipelining the bus. Latching is expensive, however, and bus failures increase with an increased number of mechanical connections. If a 16ns cycle is not a reasonable assumption then a 32B bus may be required.

7.3 File Adaptive Pre-Paging

Sequential files are pre-paged in the BIP MP. This is accomplished by a ***psuedo-fault*** (section 4.5) interrupt generated by the MsC when it maps into a page belonging to a sequential file and MsDIR.REF = 0. The REF flag indicates that the page has not been successfully referenced since it was brought in from Bs. In other words the Ms page fault handler resets REF in an I/O buffer frame inputting a sequential file page. The MsC automatically sets the flag on the first access, after it generates the interrupt. The MsC sends the target virtual address to the supervisor.

The Ms fault handler services the interrupt by first accessing SDT.header.SEQ.FOB of the target to determine if the sequential file is being scanned in forward or reverse order. VA__next and VA__last, the next-to-be-referenced and the last-referenced virtual page addresses are calculated from the target SID and

SDT.header.SEQ.PPM. PPM is the pre-paging marker and holds the next to be read page number relative to the segment. VA__last is mapped into the Ms map to get MsA__last. The target VA is then mapped into the Ms map to get the target MsA. This is used to reference the link header of the target page. Each sequential file page has a header primarily containing pointers (physical addresses) to the previous and next pages. The pointer to the next page, BsA__next, is determined. The Ms map is then updated, setting up the MsA__last frame as an I/O buffer. The I/O manager is then invoked to initiate I/O from BsA__next to MsA__last. If the target page is the last page in the file, no action is taken.

Direct access of a sequential file is disallowed because it presents the possibility of multiple copies of the same data object in shared memory. Consider a process reading a sequential file in the forward direction. Assume the current page is P_i. If an out-of-sequence reference is made to page P_{i-1}, which will no longer be in shared memory, a real fault will result. P_{i-1} will be paged in and on the first successful access, P_i will be pre-paged in, creating two copies of P_i. The multiple copies can be avoided by checking the Ms map before pre-paging, which takes too much time, or by not servicing a real fault interrupt caused by a sequential file reference. The latter policy is proposed and the referencing transaction should be ABORTed.

The pre-paging scheme effectively implements a single page buffer when manipulating sequential files. A two page buffer, for instance, can be created by an explicit input request for two pages before file manipulation. The calculation of the next-to-be-referenced page is the target page offset by two instead of one. When the first page is initially referenced, the third page is read; when the second page is initially referenced, the fourth page is read and so on. This variable size buffer can be implemented by an additional SDT field, SDT.header.SEQ.BS, the buffer size.

8. Lock Management

This section contains a brief overview of the lock mechanisms used in the shared memory manager to support ***level-3 consistency.*** Data operations within several transactions are level-3 consistent if the result of processing them is identical to the result of executing the transactions in any particular serial order (GRAY78).

8.1 Locking Protocol

A hierarchical lock structure is proposed:

```
segment
   |
page (64KB)
   |
line (4KB)
```

The locking hierarchy is an abstract level of address space partitioning above the virtual address space. Whereas the virtual space has a minimum addressing granularity of one word, the lock space has a minimum ***locking granularity*** of 4KB, called a ***lock line.*** Locking a data object grants certain access rights to the lock holder. The three tier lock granularity is considered a reasonable trade-off between concurrency permitted over a wide range of applications and implementation cost. Varying granularities is suggested in (RIES79) and seems to be justified, especially in a system with a large virtual page size.

The limiting constraints on the lock hierarchy are cost of implementation and performance degradation. The memory hierarchy must allow transfers of the lock line *at all levels.* A 4KB lock line is suggested because the burst multiplexor channel design permits this. I/O efficiency is reduced and channel device complexity increased if the lock line falls below 1 - 4KB. Increased concurrency could be gained with a 1KB lock line; however, this is costly to implement because the Ms line size is 4KB, i.e. the shared memory map keeps track of data objects of 4KB granularity.

A generalized, upgradable lock set with ***explicit*** and ***intentional*** locks is assumed (GRAY78). There are six initial ***lock types:*** N, IS, IX, SIX, S, and X. Explicit locks, S and X, lock a data object ***shared*** or ***exclusive***, respectively. Intentional locks, IS, IX and SIX, indicate an intent to lock on a smaller granularity within the target object. N is a ***nil*** lock type. Note a lock line can only be explicitly locked, whereas locks higher up in the hierarchy can be locked either implicitly or explicitly. The corresponding ***compatibility*** and ***maximum functions***, Fc and Fm, are given in (NAUM79). Fc is used to determine if a requested lock can be granted, considering the locks already held on the target data object. Fm is used to determine the equivalent or maximum lock type of a group of compatible lock types such that any lock request compatible with the maximum is compatible with all the currently held locks.

The primary instructions supported are:

```
LOCK(TID,lock_type,lock_size,VA,wait)
UNLOCK(TID,lock_size,VA)
RELEASE(TID)
```

LOCK is used to request a lock. If the wait flag is set, the transaction indicates its intent to wait until the lock is granted. If the request is granted, indicated by a successful return code, the lock is considered held by the transaction, i.e. the transaction is appended to the ***granted group*** of the data object. If the transaction intends to wait and is denied the lock, the lock manager invokes the scheduler to switch out that transaction. The transaction is appended to the ***waiting group*** of the data object. Waiting group reactivation scheduling issues are not dealt with in this thesis. Essentially whenever a lock is released, the potential exists for a waiting group transaction to move to the granted group. The overhead in performing this check as well as the complexities of the waiting group queueing discipline present problems.

UNLOCK is used to release shared locks on R/O files before transaction end, however once a lock is released, it should not be requested again or level-3 consistency is not guaranteed. In other words, shared locks can be granted in normal two-phase (NAUM79). For recovery implementation reasons (section 10), all exclusive locks must be released at one time at transaction end. This is performed by the RELEASE procedure.

The processors in the BIP MP all route lock requests to the *lock manager* running on the supervisor. These requests are serialized, making them consistent in the sense that the lock manager always sees an updated version of the lock maps because it is the only process with R/W access to the lock maps. Other supervisor procedures, for instance COMMIT and ABORT, have R/O access.

Transaction references proceed with no check whether they hold a lock on the referenced data object. Consistency is entrusted to the integrity and accuracy of the user or compiler produced locks within a program. Assuming all LOCK, UNLOCK and RELEASE commands are properly placed as defined above, a processor will not receive a copy of an object unless it has set intentional locks on the parents of the object and set an explicit lock on the object itself. Thus no other processor can access that object if it is held with an exclusive lock. If a reference is not permitted no copy of the referenced page is permitted in the private memory of the unsuccessful transaction's processor. Therefore *only multiple copies of shared objects exist between processors.* This is acceptable because no transaction holding a shared lock in the granted group for that object will be able to convert its lock to a lock incompatible with the other lock holders, e.g. a common case is an attempted S to X conversion.

In general, the lock manager prevents incompatible conversions by checking the requested lock type against the maximum lock type of the granted group with the compatibility function. The conversion can be granted only when the two are compatible. For a conversion to an exclusive lock, only when all multiple copies of the object have been invalidated can the lock be granted and the reference permitted. Again, scheduling issues are complex so that a conversion may not be permitted even if it is legal (GRAY78).

8.2 Implementation

The lock manager, performing a task similar to a data base manager, keeps a ***consistency list,*** CLIST, which is a cross-referenced lock map (fig. 8.1). The CLIST is a complex pointer structure because the sizes of the granted and waiting groups for each level of the lock hierarchy vary. There are two types of CLIST entries: ***level*** and ***group.*** There are three types of level entries: ***file, page,*** and ***line,*** and two types of group entries: ***granted*** and ***waiting.*** To each of these entry types, except for the file entry, there corresponds a chain type of the same name.

The CLIST is entered by an indirect pointer in the target SDT entry for both LOCK and UNLOCK procedures. The following is a description of the LOCKing sequence. SDT.header.FPT points to the file entry corresponding to the target segment. If SDT.header.FLV = 1, the file entry is valid and is then accessed. The search continues if the requested lock size is smaller than a segment and the requested lock type is compatible with the maximum file lock of the target entry. The search stops at the file level if a file lock was requested or if the requested lock cannot be granted. These cases are discussed below.

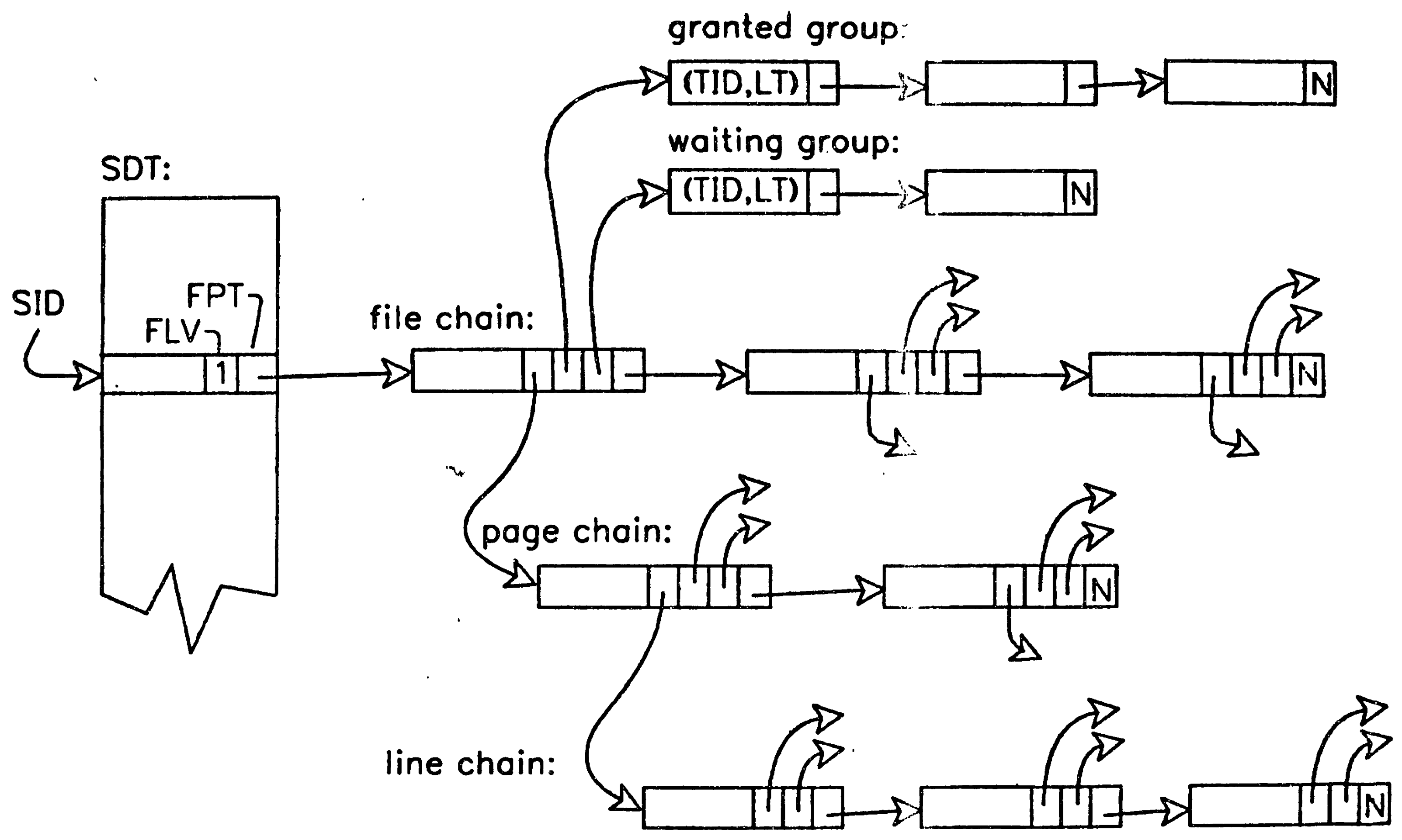


Figure 8.1. Consistency List (forward ptrs only)

If the search continues, the *page chain*, pointed to by the file entry, is then scanned sequentially. If the target page link is found, the search continues if the requested lock size is smaller than a page and the requested lock type is compatible with the maximum file lock of the target entry.

The final chain searched is the *line chain*, pointed to by the target page entry. The line chain is also searched sequentially. If the target link is found, the lock is granted or denied as previously described.

A level entry contains two pointers to a *granted group chain* and *waiting group chain*. A group chain entry holds the lock type and transaction identifier of the request. This (LT,TID) pair is interpreted with respect to its chain, i.e. a held, valid lock or a waiting lock request. Group entries containing the same TID (accessed by the same transaction) are linked in a circular *transaction chain*. The anchors of these chains are kept in a transaction anchor table, TAT, addressed directly by TID. The CLIST structure can thus be cross-referenced by TID or VA, for efficient implementation of the LOCK, UNLOCK, RELEASE, COMMIT, and ABORT procedures.

The CLIST search is ended by one of three conditions: no level entry exists for the data object specified in the request, the request is denied by a higher lock or the target level entry is found. The first condition is dealt with by creating a new entry, linking it to the proper level chain, creating a single link granted group containing the (LT,TID) pair and finally linking that group entry to the proper TID chain. The second condition causes the request to be appended to the level entry's waiting group if the request so specified (wait = 1). The third condition is dealt with as previously described.

The UNLOCK sequence is similar to the one outlined above, except that the lock is always released if it is not exclusive. The lock is released by unlinking it from its granted group and updating the maximum lock in the corresponding level entry.

The RELEASE procedure accesses the CLIST via the TAT. The target transaction pool is scanned and each link is essentially UNLOCKed. The recalculation of the maximum held lock at each invocation of RELEASE is time consuming but is suggested to permit greater concurrency.

9. Transaction Management

The COMMIT and ABORT procedures are the major parts of the **transaction manager**, a member of the global operating system.

9.1 Commit

A transaction commits when it is finished processing, by executing the COMMIT procedure (fig. 9.1). This implies a single ***commit point*** at transaction end using the notation in (NAUM79). It is assumed that the transaction is switched out when the commit is requested because the procedure is lengthy.

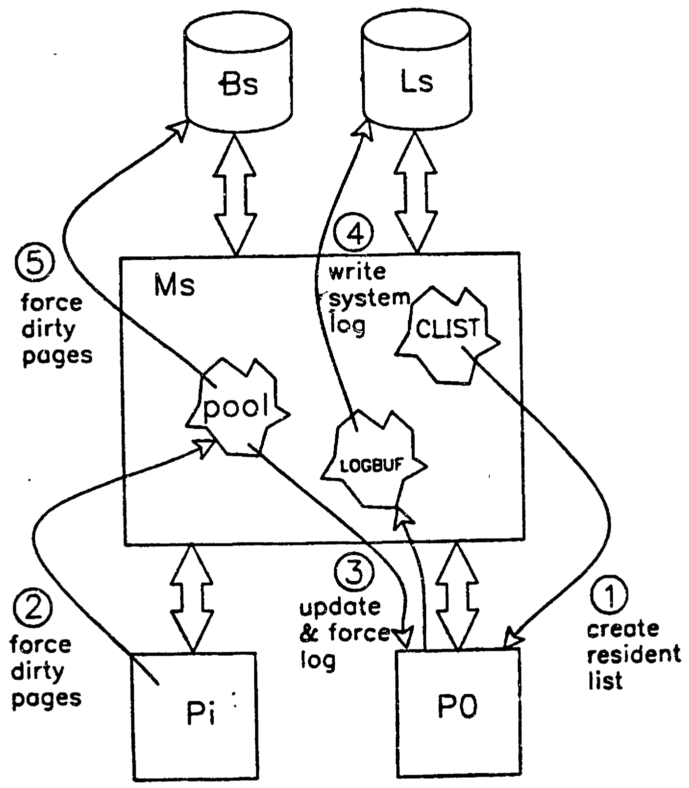


Figure 9.1. COMMIT procedure

The purpose of the commit is to output all the modifications made by the finished transaction to nonvolatile memory. Multiple copies of data objects at different levels in the memory hierarchy and in various states of update must be accounted for. In the IBM 370/168 where the cache is not architected, an output request triggers a cache interrogation, in hardware, to check for multiple copies. The interrogate function is integrated with the actual data transfer to channel. In the proposed system, an L_1 and L2 interrogate, using INTL PID, is necessary in processor PID running the transaction. This forces multiple copies of the data objects up to Ms. This interrogate cannot be integrated with the channel transfer because first the log [1] must be written. After the interrogate, the log can be written by simply referencing the dirty objects by their virtual addresses. The log is forced to the log buffer, LOGBUF, in shared memory by interrogating the supervisor for the LOGBUF segment. LOGBUF is then written to the nonvolatile log store Ls. The final steps are writing back the modified data objects to Bs and then releasing all locks held by the committed transaction. The COMMIT procedure is outlined below.

1. Read in the Ms map process pool of the committing transaction. Note that the Ms map entries read are automatically locked by the MLL instruction (section 3). Enter these map entries, their indices (Ms frame numbers) and their corresponding virtual addresses (generated by scanning the hash chain) into a ***resident list***. The hash chain entries scanned must be unlocked with the MU instruction (section 3).

2. Interrogate the committing processor private memory for all Ms line copies. These interrogates are generated by examining the MC vector and issuing INTL instructions for 4KB virtual addresses corresponding to MC = 1.

3. Invalidate all resident list pages belonging to temporary segments and add these frames to the free pool using the MSU instruction. Remove these entries from resident list.

4. Append the log with REDO entries for all resident list entries with DL = 1. A REDO entry is formed by referencing the modified 1KB line by its virtual address.

[1] All forward references to logging are defined in section 10.3.

5. Write P1__COMMIT entry into log.

6. FORCE__LOG, causing the log to be forced up to shared memory LOGBUF.

7. WRITE__LOG, causing LOGBUF to be written in the nonvolatile system log.

8. Write-back all resident list modified 1KB lines to Bs by first setting MsDIR.IOF with the MSU instruction and then invoking the I/O manager to initiate I/O. The first action lessens Ms map contention by unlocking all map entries during write-back. The MsC can therefore operate unrestricted during during a possibly long I/O sequence. Any reference to an I/O locked page, IOF = 1, will cause an interrupt to the supervisor (section 4.5).

9. On all successful I/O, update the committing process pool map entries by resetting MsDIR.VE, invalidating the page frames, and linking them to the free pool, using the MSU instruction.

10. Write P2__COMMIT entry into system log. This may be implemented as a special case of I/O and hardwired directly into the channel device servicing Ls. Otherwise the log is written in the manner described in steps 5 - 7.

11. RELEASE the committing transaction's locks.

9.2 Abort

A transaction abort purges all traces of the target transaction, i.e. all modifications made in memory. The ABORT procedure is outlined below (fig. 9.2).

1. Read in the Ms map process pool of the committing transaction. Enter these map entries, their indices (Ms frame numbers) and their corresponding virtual addresses (generated by scanning the hash chain) into a resident list. The hash chain entries scanned must be unlocked with the MU instruction.

2. Interrogate the aborting processor private memory for all Ms line copies. These interrogates are generated by examining the MC vector and issuing INTL instructions for 4KB virtual addresses corresponding to MC = 1.

3. Invalidate all resident list pages and add these frames to the free pool using the MSU instruction.

4. Write ABORT entry into system log in manner similar to COMMIT step 10.

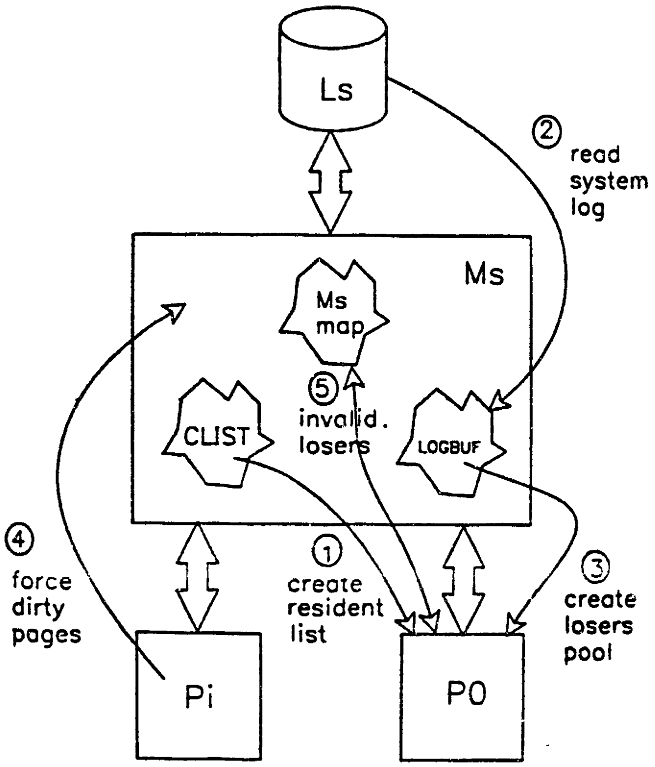


Figure 9.2. ABORT procedure

10. Recovery Management

The BIP MP recovery system incorporates a hybrid hardware and software mechanism to return the system memories to consistent states after memory failure. The recovery schemes described in this thesis are high level procedures implemented in software. The issues of the low level hardware recovery, which is essentially the function of detecting a bad memory module, are not dealt with. Software induced system failures and non-volatile memory failures are not discussed either.

The idea behind a recovery mechanism is to prevent volatile memory failures from having catastrophic results, i.e. bringing down the system for long time periods and necessitating a cold start. A fully recoverable system need cold start only once in its lifetime, although it may need to recover often (GRAY78). Only L2 and Ms module failures are considered in this thesis. L1 is a 4-way set associative cache so that L1 module failure would prevent addressing sections of the virtual space. It is assumed that if L1 crashes, the associated processor is removed from the system until L1 is fully operational.

After a crash, it is assumed that an automatic low level recovery mechanism determines which module has failed and bypasses the failed page frames by setting the appropriate flags in the directory entries. If a shared memory module failed, all processors all halted. If a private memory module failed, only the associated processor is halted. A failed shared memory frame is indicated by a set MsDIR.BAD flag in the corresponding Ms map entry. The level two cache has a one bit flag, NR, in each L2rs entry, for the same purpose (section 4.3).

The hardware recovery unit interrupts the supervisor, switching in the ***recovery manager***, a member of the global operating system. If the pages dedicated to the L2 map fail, the manager selects a new area for the map (three pages) and updates the L2HIT and L2DIR base registers, **L2HIT__base** and **L2DIR__base** (Appendix B) to point to these areas. All map entries are then set invalid. At this point, normal recovery procedures begin, causing demand paging when necessary. If the Ms map store fails, a backup store is chosen by the recovery manager. The backups are updated concurrently with the main map store, so that an up-to-date copy should be available. If this is not the case, all entries are invalidated, i.e. the complete state of memory is lost. Normal recovery procedures begin at this point.

The two-phase locking protocol (section 8.1) implies that transactions that are uncommitted at crash time have not made permanent modifications to the system state. Transactions which did commit have made permanent, consistent modifications to the system state. Thus if commit was an atomic procedure, recovery would simply involve re-trying the uncommitted transactions. Actually, even if commit were not an atomic procedure, re-try would be a sufficient but not necessary recovery action. The subtlety lies in the structure of commit, which is divided into two phases. It will be shown that recovery transactions that have successfully completed Phase I do not need to be re-tried but only redone. Redoing a transaction is less costly than re-trying and with the initial assumption of detectable volatile memory failure only, redoing simply involves reconstructing the contents of the failed frames only, in another area of memory. The reconstructed memory must also be written back to secondary memory to permanently record the correction.

Recovery involves resetting the system state to the last consistent state before the failure occurred. A uniprocessor state consists of cache, main memory contents and architected register contents. A mulitiprocessor system state consists of all processor states in addition to shared memory contents. A generic recovery operation involves

- marking memory which crashed as bad.
- redoing all transactions which finished Phase I of their commit but not Phase II.
- re-trying all transactions which did not reach Phase I completion.

Uncommitted transactions do not have to be undone because the system does not store-in-place. Obviously the information required by the recovery manager must be stored somewhere safe if it is not in Bs (because the unlucky transactions hadn't completed Phase II of commit). This safe place cannot be volatile memory and is called a ***log*** (section 10.2.3).

Few transactions should need redoing because the commit protocol specifies that the dirty pages should be forced to Bs. Therefore these pages will not be resident in Ms for long periods of time. This scheme does not increase disk contention because there is a complementary replacement policy of no write-back until commit. Notice also that the log is written at commit, as opposed to updating the log on each L2 to Ms transfer, L1 to L2 transfer or even on each store reference. This policy trades efficiency for recoverability. An assumption implicitly made is that transactions do not live excessively long lifetimes, so that the probability of a crash occurring within a given transaction life span is small. Therefore it matters little whether the log is written during or at the end of a transaction. The same line of reasoning shows that the usefulness of the log is to backup the transaction (Phase I commit) *within* its short life span because there is a much greater probability that a crash will occur before the transaction is deactivated (reaches Phase II completion).

The recovery scheme presented is contrasted to a general two-phase locking protocol (GRAY78). The general protocol requires a mechanism for ***undoing*** transactions, which is necessary for a store-in-place memory update policy. An undo implementation roughly doubles log size by introducing additional UNDO or "before" entries. A general protocol also may cause many transactions to be redone during recovery if its commit protocol does not specify *forcing* the dirty pages to Bs. This is included as the last step of the proposed commit protocol.

The basic idea in recovery is to keep a log of incremental changes to the system state to permit reconstructing a previous consistent state from the presently erroneous one in as little time as possible. The overhead in storing the required recovery information cannot be allowed to degrade normal system operation. The log mentioned is the ***system log*** and contains a large percentage of REDO entries. These entries contain the modified value of a 1KB line, write accessed by a transaction. No log entry is made for load references or for references to unrecoverable segments, such as temporary files. This is one reason that uncommitted transactions cannot be redone at crash time and then re-tried because their process state cannot be recovered since it is mostly contained in temporary segments. The name REDO comes from the eventual need for these entries in redoing the transaction. Note that the line size here, or ***log granularity,*** is not necessarily equivalent to the lock granularity. Log granularity must be relatively small to permit efficient implementation of the system log (so the log doesn't get too big). Lock granularity is chosen by considering the trade-offs between the concurrency desired, probability of deadlock and lock manager computation and lock map overhead.

The system log is a sequential ordering of system events. These events and their corresponding log entries considered in this thesis are:

RESTART	restart system
CHECKPT	checkpoint
START__TRANS	start transaction
P1__COMMIT	phase 1 commit complete
P2__COMMIT	phase 2 commit complete
ABORT	abort transaction
REDO	transaction write access

A RESTART record is written to the log whenever the recovery manager is invoked. A CHECKPT record is written to the log during system ***checkpointing.*** This is performed by the recovery manager periodically. The CHECKPT log entry consists primarily of an ***active transaction list*** written by the dispatcher. This list contains the TIDs of all uncommitted transactions at that time, implicitly defining all committed transactions. A START__TRANS record is written by the dispatcher whenever a transaction is switched in. P1__COMMIT and P2__COMMIT entries are written to the log by the COMMIT procedure after Phase I and Phase II completion, respectively. An ABORT record is written by the ABORT procedure after a transaction has successfully been rolled-back, i.e. all modifications made by that uncommitted transaction have been purged from volatile memory. REDO entries are written by the COMMIT procedure during Phase I.

The log is written according to a ***log update protocol.*** For conventional uniprocessor recovery, the log is a complete ordering of events, i.e. chronological. The BIP MP log is not strictly chronological because the log is not written until commit time. The log's ordering is logically consistent in the sense that scanning the log sequentially will reproduce the exact same system state as was actually produced (section 10.2). The system log can only be accessed through the ***log manager,*** a part of the recovery manager. A brief description of the system log implementation is given in section 10.3.

A multiprocessor has two levels of recovery: local and central. A central crash refers to Ms module failure and a local crash refers to a specific processor's L2 module failure. Both types of recovery are performed by the recovery manager running on the supervisor.

10.1 Central Recovery

Recovery from a shared memory crash proceeds as follows (fig. 10.1). First the automatic recovery mechanism runs. Second the recovery manager is invoked. A

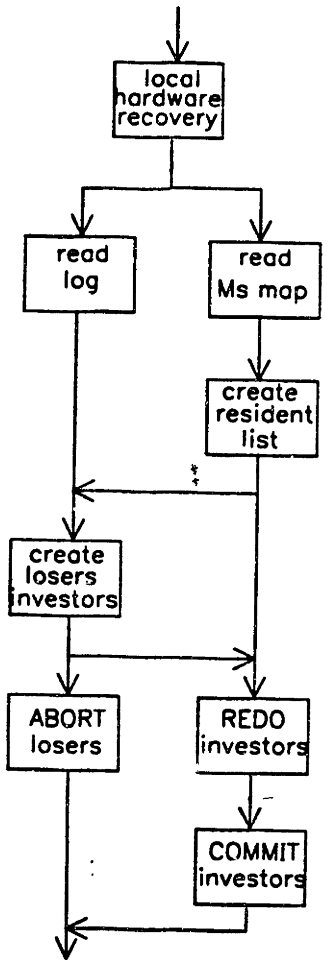


Figure 10.1. Central MP Recovery

scan of the shared memory map is made. For each entry, corresponding to a memory frame, that has been marked BAD and is still valid, the virtual address of the page resident there before the crash is calculated. The is done by traversing the MsDIR hash chain until the MsHIT entry is reached. The virtual address is simply the concatenation of MsDIR.VK of the target entry and the MsHIT index of the target hash chain. The failed frame entry is then marked invalid. Once a directory entry is marked BAD and invalid, it will become transparent to the machine. The failed virtual page address list is called the ***resident list.*** A resident list entry also contains the corresponding MsDIR.VL field, to indicate which Ms lines were present when the memory failed.

The recovery manager then invokes the log manager to read the system log since the last checkpoint. The active transaction list TIDs are entered in a ***losers pool*** (GRAY78). The log entries are then scanned sequentially from past-most to present and each time a P1__COMMIT entry is found, the corresponding TID is removed from the losers pool and added to an ***investors pool.*** The investors pool is so named because member transactions have invested in recovery insurance by writing the system log with their modifications. When a P2__COMMIT is found, the corresponding TID is removed from the investors pool. All losers are ABORTed and then added to the scheduler's queue.

No data object in the resident list which is also a transaction pool member of an investor can be a parent. In other words no investor transaction can have data objects in a failed shared memory module which reside *below* the shared memory level. This can be proved by contradiction. Assume a copy of the investor object existed on the private memory level. Then it must have gotten there after that investor's Phase I commit because Phase I specifies forcing all referenced data objects to Ms. In addition, it must have gotten there by another transaction's reference, since by definition, the investor is waiting for its Phase II commit to be completed and all user transactions commit at transaction end. This new transaction would be a loser because if it had reached Phase I completion, the data object in question would have been forced to Ms. Since the losers were previously ABORTed, that object could not possibly exist in memory at all! Therefore a contradiction is found in the initial assumption. This proof is useful in showing that the processors do not have to be interrogated over the investor pool, as was done in the ABORT for the losers pool.

In the next stage of recovery, the recovery manager scans the log again. For each REDO entry, it checks if the REDO TID is a member of the investors pool. For a match, it checks if the VPA of the modified REDO line is in the resident list. For a match, the REDO line is stored into that virtual line. This will demand page in the old copy of the virtual page, updating it. For no match, nothing is done, i.e. the REDO line was not present in the failed memory so no reconstruction is needed. When the scan is complete the recovery manager invokes Phase II of commit for each investor.

10.2 Local Recovery

Recovery from a local crash requires only the processor in question to halt. The recovery is run by the recovery manager. No reconstruction is needed because the crash cannot destroy recoverable information. This is because the commit procedure forced all modified data objects up to shared memory during Phase I. Thus dirty objects lost in an L2 crash will not belong to Phase I committed transactions. All the recovery manager can do is ABORT and then retry all losers.

The recovery manager creates a resident list (section 10.1) from the L2DIR of the crashed processor. It references the map by first forcing it up to Ms by interrogating the L2 map segment on the crashed processor. The system log is read from the last checkpoint and the manager scans the resident list, creating only a losers pool. The losers are ABORTed and then added to the scheduler's queue.

Local recovery requires access to the L2 map by the supervisor to determine which transactions were effected by the memory failure. L2 map reference by the recovery manager must be preceded by forcing the the map from the processor's private memory to shared memory for consistency. The L2c must request a transfer of the L2 map from Ms after local recovery and before attempting an L2 mapping.

When the private memory of the supervisor fails, another processor is chosen to house the global operating system and the recovery manager runs on this other processor.

10.3 System Log

The system log is implemented as a hierarchy.

$$\begin{tikzcd}
\text{Ls} \arrow[d, dash] \\
\text{system log buffer} \arrow[d, dash] \\
\text{LOGBUF}
\end{tikzcd}$$

Ls is the highest level of the hierarchy and the most secure. It is implemented as an nonvolatile magnetic store, probably disk. In practice multiple copies of Ls will be kept on separate devices to permit recovery from disk failure. The bottom level of the hierarchy, LOGBUF, is the shared memory system log area, a frozen set of page frames in Ms dedicated to buffering system log I/O. This I/O is performed by the log manager. The ***system log buffer*** is a nonvolatile electronic store, implemented to permit fast off-load from LOGBUF to Ls (COCK81). The system log hierarchy is transparent to the supervisor, which uses the log manager as an access method to the log. The system log buffer is transparent to the log manager, which simply performs I/O to an Ls device. The design issues of the system log buffer will not be discussed and therefore the upper two levels of the hierarchy will be referred to as Ls.

LOGBUF is written only in supervisor state during Phase I commit. It is assumed that most supervisor procedures are transactions themselves and therefore the system log backs-up their actions (of course this cannot be the case for the log manager). LOGBUF is forced to Ls at Phase I completion of commit. The structure of the commit protocol forces dirty objects up the memory hierarchy so that they can be logged. If LOGBUF fills up before all REDO entries are entered, it is simply forced to Ls and cleared. The P1__COMMIT entry is written to LOGBUF only when all modified lines are entered as REDO entries. Then the log manager is invoked to write the LOGBUF out to Ls. If a crash occurs before a transaction's P1__COMMIT log entry has been transmitted to Ls, that transaction is a loser by the recovery protocol and will have to be re-tried.

Logging is synchronized with committing. Thus the question whether the log is "in the right order", i.e. will produce the correct consistent state on recovery, reduces to whether the commits are serviced "in the right order". This is not a problem because asynchronously arriving commit requests are serviced one at a time, to completion, by the transaction manager. By the locking protocol, a transaction waiting for commit will still hold all its locks, in addition, no two transactions will be committed concurrently, and so the system remains consistent no matter how long a commit takes.

The log manager has three primary operations:

```
READ__LOG(first__pt,last__pt)
WRITE__LOG(entry__type,info)
FORCE__LOG
```

The supervisor uses READ__LOG as an access method to Ls, reading all records between the given indices. FORCE__LOG forces the log up from the supervisor's private memory to LOGBUF, by interrogating the supervisor for the system log segment. The LOGBUF frames define a special pool in the Ms map. This pool is scanned to determine the proper 4KB objects to interrogate. WRITE__LOG is an an output access method for writing LOGBUF to Ls.

11. Summary and Future Work

This thesis is an organizational overview and feasibility study of a memory hierarchy for a very high performance multiprocessor. The overview includes both hardware and software descriptions of relevant algorithms. These descriptions comprise the structure of the memory system and the operating system supporting the hardware. It was necessary to carefully match the architecture (user's view) and organization (designer's view) of the memory hierarchy to meet the system requirements in a clean manner.

The preliminary investigation used a queueing network model to show the feasibility of meeting the performance requirement in a scientific batch environment. The memory organization modeled consisted of a shared memory and private processor memories. The major points derived from this model were:

- Advanced disk technology *is* necessary because fast (less than 10ms) mean disk arm access time is crucial.
- A low shared memory miss ratio (less than 0.075) is necessary.
- A fast system bus (close to 1GBS) is necessary.

The above requirements define a well-balanced system. It was shown that by meeting these constraints, the degree of multiprogramming and number of simultaneously active disk arms become non-limiting parameters.

A memory hierarchy was outlined, concentrating on the level two cache, shared memory and I/O channel device. The feasibility of a hardware controlled fully-associative 4MB L2 cache was shown. Mapping and replacement hardware, utilizing an "inverted" page table and a bit scanning algorithm respectively, were described.

An L1-L2 cache interface was defined, where L1, the level one cache, was assumed to be similar to an IBM 370/3033 cache.

The shared memory, Ms, organization entails a hardware mapping unit, similar to the level two cache controller, and a software replacement manager. The Ms map and page frame access counts (for the replacement algorithm) are kept in a dedicated store because of inefficiencies in a decentralized organization. Entries in this store are locked in hardware to prevent inconsistent updates by the mapping and replacement units. An elaborate interrogation protocol was described to solve the problems of multiple copies and orphans.

A burst-multiplexor channel device was described for servicing a group of disks in a cost/performance efficient manner. In non-degraded mode, the device services a maximum of 32 asynchronous disk arms, performing serial to parallel data stream conversion and frame buffering for each. The buffered data frames are burst to the shared memory on the system bus at 1GBS.

An outline was given of the global operating system managers supporting the memory hierarchy. These included the Ms page fault handler (mentioned above) and the file, I/O, lock, transaction and recovery managers. The latter three perform optional database environment functions. A description was given of a streamlined two-phase locking procedure and the related commit and abort procedures. The hierarchical locking scheme was shown to guarantee level-3 consistency in a multi-level memory MP dependent on user (compiler) integrity. The lock granularity was fitted to the memory organization for efficiency. Both private and shared memory failure recovery procedures were summarized, built upon a log update protocol tailored to the locking procedure (no UNDO records).

In summary, this thesis concentrates on the memory hierarchy organization and support for a very high performance multiprocessor. The study found no major technical problem with the design, for both scientific and database applications, and provided a set of general criteria for a well-balanced system.

The following is a list of research areas to be explored concerning the BIP MP.

- Modeling system failures on all levels and the design of necessary safeguards to make the BIP MP ***fault tolerant.*** (KING81) describes an analytic model used to evaluate independent processor failures. Extensions are needed to model shared memory and channel device failures.

- The log hierarchy implementation for the database system. The log granularity, specified here as 1KB may be unreasonably large. The log update protocol is expected to be a major bottleneck to system performance (COCK81).

- Implementation of a 1GBS multi-user bus.

- Design of the BIP MP operating system. This may entail partitioning the scheduler and/or dispatcher among the processors, in which case a message passing facility must be developed. The interrupt sequences of the system must be defined and supervisor code structured to operate efficiently. The lock, recovery, file, I/O and transaction managers must be prototyped and tested to determine bottlenecks.

- The L2 and Ms mapping and replacement algorithms simulation on a fairly low level with application reference traces.

- Examine organizations, support and cost/performance trade-offs of additional memory levels, such as ***disk cache*** (TOKU80) and ***mass store*** (archive storage).

- Design of a ***multi-port*** shared memory with multiple system buses in the high performance environment at a cost/performance advantage.

Appendix A : Technology

Technology assumptions are categorized as logic, volatile and nonvolatile memory. Packaging and mass store technologies are not pertinent to this paper.

Logic

Logic, which includes registers, is assumed to be implemented in ***gate array*** technologies (POSA80, HART81). Three specific product lines were examined: Fairchild F300 ECL (HIVE78), Motorola MCA MECL (PRIO) and Texas Instruments TAT Schottky (TI80). Top of the line arrays in these families approach > 2000 gates and < 1ns gate delays with predictions for greater density in five years. The arrays are normally organized in macro cells. Assumptions here are for 100% routability and utilization. The following list summarizes model technology restrictions.

# gates	= 2000
gate delay	= 1ns
#I/O	= 120
max #O	= 60
max fanin/gate	= 4
max fanout	= 4
max emitter-dotting	= 8 (wired-OR)
# phases/gate	= 2 (NOR/OR)

Arrays

Volatile memories are designed around standard currently available RAMs (POSA80, BURS81). These are summarized below.

level	form	access	technology
L1	1Kx8	10ns	BIPOLAR-ECL
L2	1Kx8 64x8	50ns	FET MOS (static)
Ms	256Kx1	400ns	FET MOS (dynamic)

Selection the dynamic RAM for the shared memory is a complex decision. Faster RAMs at the same density are predicted in five years. In general, the packaged access time of the memory is calculated as a package delay and address decode in addition to the chip access time. It is assumed that the refresh penalty of dynamic RAM will not effect system performance. If the packaged access time of the shared memory could be reduced by 50% then twice the number of modules could be configured for the same sized memory. This buys increased reliability. If the packaged access time is reduced by only 25%, then twice the number of modules implies increasing Ms size by 50%. Reliability has been bought along with increased hit ratio for twice the price. The decision can only be made knowing the packaging technology.

Disk

Involatile memory is assumed to be disk. Current moving arm technologies, e.g. IBM 3380, have rotational transfer rates of 3MBS (BRAN81b). Predictions are for double linear density (to 1.2 GB/arm) in five years, implying 6MBS transfer rates. The term ***disk arm*** as used in this paper refers to a single movable structure with several read/write heads (one per platter surface) capable of delivering a single data stream at the rate of Ract = 6MBS.

The 3380 has an average seek time of 16ms (BRAN81b), although it is shown in (JONE81a) that a substantial percentage of I/O requests require no seek. The 3380 latency is 16.7ms for a full revolution, implying a mean rotational delay of about 8ms.

Methodology

The design methodology is IBM **Level Sensitive Scan Design,** LSSD (BERG79). This essentially provides a means of testing the hardware at the cost of additional gates and I/O pins. The basic ***shift register latch,*** SRL, is used in the shift chip design and all registers. A scan ring facility is thereby provided for testing. It also permits a two phase system clock which is assumed in the L2 design. The BMC design assumes that the I/O device control units issue a two-phase asynchronous clock with their serial data transmission.

Methodology

The design methodology is IBM Level Sensitive Scan Design, LSSD (BERG79). This essentially provides a means of testing the hardware at the cost of additional gates and I/O pins. The basic ***shift register latch,*** SRL, is used in the shift chip design and all registers. A scan ring facility is thereby provided for testing. It also permits a two phase system clock which is assumed in the L2 design. The BMC design assumes that the I/O device control units issue a two-phase asynchronous clock with their serial data transmission.

Appendix B : L2 Specification

The hardware designs in this paper are given in a hybrid HEX/PASCAL language specification. HEX is a hardware description description language (DENN81) which can be accompanied by a dataflow for clarity. Minor modifications in the language was introduced to make the description clearer and more succinct. The PASCAL elements play the role of assembly directives. Because HEX does not permit iterative actions, recursive ROUTINEs were used to produce the same behavior.

The use of HEX in this paper should not be construed as an attempt to fully specify a working hardware system. It is rather used as a descriptive aid in explaining the control algorithms in more detailed and structured manner than plain English. Obscuring details such as parity checking and generation, are left out of both the discussion and the HEX. Unfortunately the HEX language cannot convey two abstractions which are important in the design process: chip partitioning and synchronous timing sequences. These are meant to be hidden from the designer by the HEX compiler, however since such a facility was not available, it was necessary to roughly check the realizability of the design by hand.

Chip partitioning was done for critical areas, such as the shift chip (fig. 10), by designing the hardware to the gate level. This was also done for other pieces of hardware, such as L2r, however such circuit diagrams add little to understanding the motivations and justifications of the design. Timing constraints were to first order a function of the number of memory accesses necessary for a given operation., such as L2 map. This often entailed interleaving the memories and assuming combinational logic is pipelined with memory accesses, i.e. standard synchronous machine operation.

The important differences between the HEX used here and standard HEX are listed below.

- "<",">" are used for curly brackets, "(",")" for square brackets and "<-" for left arrow.
- "<>" and XOR define additional relational operators: not equal and exclusive or, respectively.
- **CONNECTIONS** of arrays of systems are facilitated with the PASCAL **FOR** statement.
- Vectors and arrays of **REGISTERs** are permitted, for example:

```
REGISTER reg1<3:0>(127:0);      ! vector of 4
                                ! 16B registers
REGISTER reg2<1:0,3:0>(63:0);   ! 2 x 4 array of
                                ! of 8B registers
```

- HEX naming conventions are extended as follows. Two or more **CARRIERs** can have the *same* synonym, in which case the reference is clarified by prefixing the synonym with the **CARRIER** name in PASCAL RECORD/field dot notation. For example

```
REGISTER reg3(127:0);           ! single register
   HK(20:0) IS reg3(127:107);
REGISTER reg4<0:3>(127:0);      ! register array
   HK(20:0) IS reg4(127:107);
   HKindx(3:0) IS reg4<1:2>(127:124);
```

The following are some legal references, the last two of which are equivalent:

```
reg3.HK
reg3.HK(2:0)
reg4.HK
reg4<2>.HK(20:17)
reg4<2>.HKindx
```

- X is used as "don't care" when specifying a binary sequence.
- PASCAL **WITH** statement operates on synonyms. The **ENDWITH** statement is introduced as a delimiter.

- A reverse field is denoted by reversing the field subscripts, i.e. the first is less than the second. For example reg3(0:127) is the bit reversal of reg3.

```
SYSTEM L2 (N)                          ! level-2 cache
                                       ! N = 2exp7-1 bus width

    OUTLET   bus_out(N:0),             ! transfer to Ms
             L1_dlg_out(2:0),          ! L1 dialog
                                       ! 000 L2 hit
                                       ! 001 L2 miss
                                       ! 010 transfer started
                                       ! 011 transfer completed
             line_out(N:0),            ! transfer to L1
             Ms_dlg_out(1:0),          ! Ms dialog
                                       ! 00 do Ms map
                                       ! 01 do L2 to Ms transfer
                                       !    do Ms to L2 transfer:
                                       ! 10 last object mapped
                                       ! 11 specified MsA
             Ms_addr_out(27:0),        ! addr to Ms: VPA or MsA
             L2DLAT_VE,                ! L2DLAT entry to L1
             L2DLAT_TAG(9:0),
             L2DLAT_PK(2:0),
             L2DLAT_VL(3:0),
             L2DLAT_VK(24:0),
             L1_req_out(1:0),          ! L1 dialog
                                       ! 00 load L2DLAT entry
                                       ! 01 load L1DIR entry
                                       ! 10 req L1 mapping
             L1DIR_addr(5:0),          ! L1DIR index
             L2DLAT_addr(6:0),         ! L2DLAT index
             L1_addr(?:0);             ! transfer address to L1
    INLET     VA_in(43:0),          ! virtual address from CPU
              L1_status_in,         ! L1 mapping status
                                    ! 0X L2DLAT miss
                                    ! 10 L1 miss
                                    ! 11 L1 hit
              L1_dlg_in(1:0),       ! L1 dialog
                                    ! 00 req L2 map
                                    ! 01 req L1 to L2 transfer
                                    ! 10 req L2 to L1 transfer
              bus_in(N:0),          ! transfer from Ms
              Ms_dlg_in(1:0),       ! Ms dialog
                                    ! 00 Ms hit
                                    ! 01 Ms miss
                                    ! 10 transfer to L2 started
                                    ! 11 transfer to L2 completed
              line_in(N:0);         ! transfer from L1

COMPONENT L2c          ! control unit and store

    OUTLET bus_out(N:0),            ! data ti ansfer to bus
           line_out(N:0),           ! data transfer to L1
           L1_addr(?:0),            ! transfer address to L1
           Ms_addr_out(27:0),       ! addr to Ms: VPA or MsA
           L1_dlg_out(2:0),         ! L1 dialog
           Ms_dlg_out(1:0),         ! Ms dialog
           L2r_req(2:0),            ! request for L2r
           L2r_addr(9:0),           ! access address
           L2DLAT_VE,               ! L2DLAT entry to L1
           L2DLAT_TAG(9:0),
           L2DLAT_PK(2:0),
           L2DLAT_VL(3:0),
           L2DLAT_VK(24:0);
           L1_req_out(1:0),         ! L1 dialog
           L1DIR_addr(5:0),         ! L1DIR index
           L2DLAT_addr(6:0),        ! L2DLAT index

    INLET virt_addr(43:0),          ! virtual address from CPU
          L1_dlg_in(1:0),           ! L1 dialog
          L1_status_in,             ! L1 mapping status
          Ms_dlg_in(1:0),           ! Ms dialog
          line_in(N:0),             ! data transfer from L1
          bus_in(N:0),              ! data transfer from bus
          L2r_choice(9:0);          ! rep choice from L2r

ENDCOMP      ! L2c
```

```
COMPONENT L2r    ! replacement unit and access count store

   OUTLET L2LRU(9:0);        ! top of replacement queue

   INLET  req(2:0);          ! L2r request:
                             ! 000 update access count
                             ! 001 freeze frame
                             ! 010 unfreeze frame
                             ! 011 permanently freeze frame
                             ! 100 reset L2rs
                             ! 101 send L2LRU
                             ! 110 sweep
                             ! 111 search (default)
          addr(9:0);         ! L2 frame address for update

ENDCOMP     ! L2r

CONNECTIONS

L2c.Ms_dlg_out -> L2.Ms_dlg_out; L2c.DLAT_VE -> L2.DLAT_VE;
L2c.L1_dlg_out -> L2.L1_dlg_out; L2c.DLAT_VK -> L2.DLAT_VK;
L2.Ms_dlg_in -> L2c.Ms_dlg_in;   L2c.bus_out -> L2.bus_out;
L2.L1_dlg_in -> L2c.L1_dlg_in;   L2.VA_in -> L2c.virt_addr;
L2r.L2LRU -> L2c.L2r_choice;     L2.line_in -> L2c.line_in;
L2c.line_out -> L2.line_out;     L2c.L1_addr -> L2.L1_addr;
L2c.DLAT_PK -> L2.DLAT_PK;       L2c.DLAT_VL -> L2.DLAT_VL;
L2c.DLAT_TAG -> L2.DLAT_TAG;     L2c.L2r_addr -> L2r.addr;
L2c.L1DIR_addr -> L2.L1DIR_addr; L2.bus_in -> L2c.bus_in;
L2c.L1_req_out -> L2.L1_req_out; L2c.L2r_req -> L2r.req;
L2.L1_status_in -> L2c.L1_status_in;
L2c.L2DLAT_addr -> L2.L2DLAT_addr;
L2c.Ms_addr_out -> L2.Ms_addr_out;

ENDSYS      ! L2
```

```
SYSTEM L2c          ! L2 controller

CARRIERS      ! L2c

! L2 store

MEMORY mbank0<0:2exp16-1>(N:0),    ! 4MB L2 store
       mbank1<0:2exp16-1>(N:0),
       mbank2<0:2exp16-1>(N:0),
       mbank3<0:2exp16-1>(N:0);

REGISTER addr_reg(20:0),            ! L2 store address reg
     bnkno IS addr_reg(20:19);      ! memory bank no.
     indx  IS addr_reg(2:0);        ! index of L2HIT entry

REGISTER out_reg<0:1,0:3>(N:0),     ! L2 store output staging reg
                                   ! L2DIR entry definition
HK(20:0)  IS data_reg(63:43);       ! hash key
FPH(10:0) IS data_reg(42:32);       ! ptr into L2HIT
FPD(9:0)  IS data_reg(41:32);       ! forward ptr (entry index)
FPA(8:0)  IS data_reg(41:33);       ! forward ptr (16B index)
EOC       IS data_reg(31);          ! end of chain
VE        IS data_reg(30);          ! valid entry
VL(3:0)   IS data_reg(29:26);       ! valid line
AL(3:0)   IS data_reg(25:22);       ! arriving line
DL(3:0)   IS data_reg(21:18);       ! dirty line
PK(2:0)   IS data_reg(17:15);       ! protection key
MSA(13:0) IS data_reg(14:1);        ! MsA of parent

REGISTER in_reg<0:3>(N:0);         ! L2 store input staging reg

REGISTER L2HIT_reg(15:0),     ! mapping reg to hold L2_HIT entry
                             ! L2HIT entry definition
     PI(9:0) IS L2HIT_reg(15:6); ! index to L2DIR
     VE      IS L2HIT_reg(5);    ! valid entry

REGISTER count(6:0),                ! transfer counter
     bindx(1:0) IS count(1:0);
     ovr        IS count(3);        ! 3 bit count is over
     ovrflw     IS count(6);        ! 6 bit count is over

REGISTER int_count(5:0),            ! interrogate counter
   int_cnt(4:0) IS int_count(4:0);! 5 bit counter
   over         IS int_count(5);  ! 5 bit count is over

REGISTER bankwren(3:0),             ! bank enable on write
         L2HIT_base(9:0),          ! points to map frame
         L2DIR_base(8:0),          ! points to map frames
         rep_addr(9:0),            ! holds LRU rplcmnt choice
         rep_ptr(10:0),            ! holds L2DIR entry index
                                   ! pointed to by rep entry
         trnsfr_addr(9:0);         ! holds L2A of 4KB from Ms
```

```
FLIP-FLOP rden,                         ! read enable
          wren,                         ! write enable
          sel;                          ! out_reg even/odd 8B select

! buffers

MEMORY in_bbank0<0:63>(N:0),            ! 4KB buffer from bus
       in_bbank1<0:63>(N:0),
       in_bbank2<0:63>(N:0),
       in_bbank3<0:63>(N:0),

MEMORY out_bbank0<0:63>(N:0),           ! 4KB buffer to bus
       out_bbank1<0:63>(N:0),
       out_bbank2<0:63>(N:0),
       out_bbank3<0:63>(N:0),

REGISTER OBV(31:0),                     ! out buffer line valid
         IBV(3:0),                      ! in buffer line valid
         out_baddr(31:0),               ! VA of out buffer object
         out_obreg<0:3>(N:0),           ! out buffer output staging re
         out_ibreg<0:3>(N:0),           ! out buffer input staging reg
         in_obreg<0:3>(N:0),            ! in buffer output staging reg
         in_ibreg<0:3>(N:0),            ! in buffer input staging reg
         in_bankwren(3:0),              ! in buffer bank write enable
         out_bankwren(3:0);             ! out buffer bank write enable

REGISTER out_baddr_reg(7:0),            ! out buffer address reg
         out_bnkno IS out_baddr_reg(7:6);

REGISTER in_baddr_reg(7:0),             ! in buffer address reg
         in_bnkno IS in_baddr_reg(7:6);

FLIP-FLOP in_brden,                     ! in buffer read enable
          in_bwren,                     ! in buffer write enable
          out_brden,                    ! out buffer read enable
          out_bwren,                    ! out buffer write enable
          out,                          ! out buffer valid flag
          wb;                           ! L2 write-back flag

BEHAVIOR   ! L2c

main_loop();
```

```
DEFINITIONS


ROUTINE read(addr(20:0) RETURNS (N:0))

! read from L2 store

rden <- '1;
addr_reg <- addr;
CASE bnkno
    '00 : RETURN mbank0<addr_reg>;
    '01 : RETURN mbank1<addr_reg>;
    '10 : RETURN mbank2<addr_reg>;
    '11 : RETURN mbank3<addr_reg>;
ENDCASE
ENDROUTINE read


ROUTINE read_L2DIR(addr(9:0) RETURNS (N:0))

! read L2DIR entry

sel <- addr(0);
RETURN read(L2DIR_base || addr(9:1) || '000);
ENDROUTINE read_L2DIR


ROUTINE read_L2HIT(addr(10:0) RETURNS (N:0))

! read L2HIT entry and put in L2HIT_reg

LET d = read(L2HIT_base || addr) IN
    CASE indx
        '000 : L2HIT_reg <- d(15:0);
        '001 : L2HIT_reg <- d(31:16);
        '010 : L2HIT_reg <- d(47:32);
        '011 : L2HIT_reg <- d(63:48);
        '100 : L2HIT_reg <- d(79:64);
        '101 : L2HIT_reg <- d(95:80);
        '110 : L2HIT_reg <- d(111:96);
        '111 : L2HIT_reg <- d(127:112);
    ENDCASE
    RETURN d;
ENDLET
ENDROUTINE read_L2HIT
```

```
ROUTINE write (addr(20:0))

! write into L2 store

wren <- '1;
addr_reg <- addr;
CASE bnkno
    '00 : mbank0<addr_reg> <- out_ibreg<0>;
          bankwren(0) <- '1;
    '01 : mbank1<addr_reg> <- in_reg<1>;
          bankwren(1) <- '1;
    '10 : mbank2<addr_reg> <- in_reg<2>;
          bankwren(2) <- '1;
    '11 : mbank3<addr_reg> <- in_reg<3>;
          bankwren(3) <- '1;
ENDCASE
ENDROUTINE write


ROUTINE write_L2DIR(addr(9:0))

! write L2DIR entry

write(L2DIR_base || addr(9:1) || '000);
ENDROUTINE write_L2DIR


ROUTINE write_L2HIT(addr(10:0))

! write L2HIT_reg into L2HIT in L2 store

CASE indx                    ! put entry in all input regs
    '000 : in_reg(15:0)    <- L2HIT_reg;
    '001 : in_reg(31:16)   <- L2HIT_reg;
    '010 : in_reg(47:32)   <- L2HIT_reg;
    '011 : in_reg(63:48)   <- L2HIT_reg;
    '100 : in_reg(79:64)   <- L2HIT_reg;
    '101 : in_reg(95:80)   <- L2HIT_reg;
    '110 : in_reg(111:96)  <- L2HIT_reg;
    '111 : in_reg(127:112) <- L2HIT_reg;
ENDCASE
write(L2HIT_base || addr); ! but only write the correct one
ENDROUTINE write_L2HIT
```

```
ROUTINE in_bread(addr(7:0) RETURNS (N:0))

! read from in_buff

in_brden <- '1;
in_baddr_reg <- addr;
CASE bnkno
    '00 : RETURN in_bbank0<in_baddr_reg>;
    '01 : RETURN in_bbank1<in_baddr_reg>;
    '10 : RETURN in_bbank2<in_baddr_reg>;
    '11 : RETURN in_bbank3<in_baddr_reg>;
ENDCASE
ENDROUTINE in_bread


ROUTINE out_bwrite (addr(7:0))

! write into out_buffer

out_bwren <- '1;
out_baddr_reg <- addr;
CASE out_bnkno
    '00 : out_bbank0<out_baddr_reg> <- out_ibreg<0>;
          out_bankwren(0) <- '1;
    '01 : out_bbank1<out_baddr_reg> <- out_ibreg<1>;
          out_bankwren(1) <- '1;
    '10 : out_bbank2<out_baddr_reg> <- out_ibreg<2>;
          out_bankwren(2) <  '1;
    '11 : out_bbank3<out_baddr_reg> <- out_ibreg<3>;
          out_bankwren(3) <- '1;
ENDCASE
ENDROUTINE out_bwrite


ROUTINE main_loop

! L2c description assumes a queueing mechanism for
! saving requests that cannot be immediately serviced.

LET a = RECEIVE L1_dlg_in IN      ! see what L1 wants done
    CASE a
        '00 : do_L2_map();        ! L1 miss requires L2 map
        '01 : do_L1_transfer();   ! L1 to L2 transfer
        '10 : do_L2_transfer();   ! L2 to L1 transfer
    ENDCASE
ENDLET
main_loop();
ENDROUTINE main_loop
```

```
ROUTINE do_L2_map

! perform L2 map by hashing into L2DIR

LET  SID = RECEIVE virt_addr(43:29),
    RVPA = RECEIVE virt_addr(12:22), ! reverse field
    hash = (RVPA XOR SID(39:29))   IN

in_reg<bnkno> <- read_L2HIT(hash);  ! enter map thru index table
CASE L2HIT_reg.VE
    '0 : L2_pg_miss();                ! no hash chain exists
    '1 : out_reg <- read_L2DIR(PI); ! fetch two L2DIR entries
         search_chain();             ! start searching hash chain
ENDCASE ENDLET
ENDROUTINE do_L2_map



ROUTINE search_chain

! do_L2_map inner loop : hash chain search

LET  VA = RECEIVE virt_addr,
    key = RECEIVE virt_addr(43:19),
    VPA = RECEIVE virt_addr(43:16),
   line = RECEIVE virt_addr(11:10)  IN

WITH out_reg<sel,> DO
IF (HK = key) THEN                    ! L2DIR entry match
    SEND PI AT L2r_addr;              ! L2 frame accessed
    SEND '000 AT L2r_req;             ! so update L2rs
    CASE VL(line) || AL(line)
         '00 : near_miss();
         '01 : SEND VPA AT Ms_addr_out;! line already arriving
               SEND '00 AT Ms_dlg_out; ! so req Ms mapping
               await_Ms();
         '10 : L2_hit();
         '11 : error();
    ENDCASE
               ELSE no_hit_yet();
ENDIF ENDWITH ENDLET
ENDROUTINE search_chain
```

```
ROUTINE no_hit_yet

! current chain link doesn't match

WITH out_reg<sel,> DO
IF (EOC = '1) THEN
     L2_pg_miss();            ! L2 miss
           ELSE
     sel <- FPD(0);           ! if sequential chain link
                              ! simply change selector
     IF (FPA <> addr_reg(12:4)) THEN  ! if non-sequential,
        out_reg <- read_L2DIR(FPD);   ! fetch next link
     ENDIF
     search_chain();          ! continue chaining
ENDIF ENDWITH
ENDROUTINE no_hit_yet



ROUTINE L2_hit

! send new L2DLAT entry to L1

SEND out_reg<sel,>.VL AT L2DLAT_VL;
SEND out_reg<sel,>.PK AT L2DLAT_PK;
SEND              PI AT L2DLAT_TAG;
SEND              '1 AT L2DLAT_VE;
SEND             key AT L2DLAT_VK;
SEND            '000 AT L1_dlg_out;    ! signal L2 hit
ENDROUTINE L2_hit



ROUTINE near_miss

! L2 page is valid but line is missing

WITH out_reg<sel,> DO
LET line = RECEIVE virt_addr(11:10) IN
    AL(line) <- '1;              ! mark target line arriving
    in_reg <- out_reg;
    write(addr_reg);
    SEND MSA AT Ms_addr_out;   ! init Ms transfer with
    SEND '11 AT Ms_dlg_out;    ! L2DIR.MSA
    wb <- '0;                  ! indicate that no write-back
    L2_fault();                ! was done to bypass buffer
ENDLET ENDWITH
ENDROUTINE near_miss
```

```
ROUTINE L2_pg_miss

! L2DIR missed: must access Ms for page and update L2DIR

LET  VA = RECEIVE virt_addr IN
IF out AND (out_baddr = VA(43:12)) THEN
     out_to_in();                 ! target is in out buffer
ENDIF
SEND VPA AT Ms_addr_out;          ! req Ms mapping
SEND '00 AT Ms_clg_out;
SEND '101 AT L2r_req;             ! fetch rplcmnt choice
rep_addr <- RECEIVE L2r_choice;! link rplcmnt frame onto
relink();                         ! target chain
int_L1();                         ! interrogate L1
L2s_to_buff();                    ! L2 write_back
await_Ms();
ENDLET
ENDROUTINE L2_pg_miss



ROUTINE relink()

! replacement frame must be removed from its
! present chain and linked to the target chain.
! Free up the replacement frame entry first.

LET  SID = RECEIVE virt_addr(43:29),
    RVPA = RECEIVE virt_addr(12:22), ! reverse field
    hash = (RVPA XOR SID(39:29))   IN

WITH out_reg<sel,> DO
out_reg <- read_L2DIR(rep_addr);
rep_ptr <- FPH;                     ! save link addr pointed
                                    ! to by rplcmnt link
scan();                             ! remove rep link from its chain
out_reg <- read_L2DIR(rep_addr); ! get rplcmnt choice again
read_L2HIT(hash);                   ! hash into target chain again
IF (L2HIT_reg.VE = '1) THEN         ! append rep link to trgt chain
    FPD <- PI;                      ! (B)
    EOC <- '0;                      ! (C)
                        ELSE        ! or create new chain
    FPH <- hash;
    EOC <- '1;
ENDIF
in_reg <- out_reg;                  ! write back L2DIR entry
write_L2DIR(rep_addr);
PI <- rep_addr;                     ! (D)
write_L2HIT(hash);                  ! write back L2HIT entry
ENDWITH ENDLET
ENDROUTINE relink
```

```
ROUTINE scan

! scan hash chain to find the link before the replacement
! link and alter its pointer to skip over the rep link

WITH out_reg<sel,> DO
IF (EOC = '1) THEN                  ! if end of chain, get L2HIT
    read_L2HIT(FLH);                ! entry and first chain link
    out_reg <- read_L2DIR(PI);
ENDIF
IF (EOC = '1) THEN                  ! single link chain
    read_L2HIT(FPH);
    L2HIT_reg.VE <- '0;             ! invalidate chain
    write_L2HIT(FPH);
             ELSE
    IF (FPD = rep_addr) THEN        ! scan complete
        FPD <- rep_ptr;             ! (A)
        in_reg <- out_reg;
        write(addr_reg);            ! write back L2DIR entry
                       ELSE        ! continue scanning
        out_reg <- read_L2DIR(FPD);
        scan();
    ENDIF
ENDIF ENDWITH
ENDROUTINE scan


ROUTINE L2_fault

! on L2 miss/Ms hit : fetch data object
! from Ms and validate L2 map

LET line = RECEIVE virt_addr(11:10)  IN
IF wb = '1 THEN wait2();       ! Ms to in buffer then
                               ! in buffer to L2s
           ELSE wait1();       ! Ms to L2s
out_reg <- read_L2DIR(trnsfr_addr);
WITH out_reg<sel,> DO          ! update L2DIR entry
    VE <- '1·
    AL<line> <- '0;
    VL<line> <- '1;
WITHEND
in_reg <- out_reg;
write(addr_reg);
ENDLET
ENDROUTINE L2_fault
```

```
ROUTINE await_Ms

! wait for Ms mapping reply and act accordingly

LET d = RECEIVE Ms_dlg_in IN
  CASE d
    '00 : await_Ms();                   ! wait for Ms reply
    '01 : SEND '10 AT Ms_dlg_out;       ! req Ms to buffer transfer
          L2_fault();                   !
    '10 : SEND rep_addr AT L2r_addr;    ! Ms miss, so
          SEND '001 AT L2r_req;         ! freeze rep frame
  ENDCASE
ENDLET
ENDROUTINE await_Ms


ROUTINE wait1

! wait for Ms to L2s transfer completion

LET d = RECEIVE Ms_dlg_in IN
    CASE d
        '0X : wait1();
        '10 : wait1();
    ENDCASE
ENDLET
ENDROUTINE wait1


ROUTINE wait2

! wait for Ms to buffer transfer completion
! then transfer buffer to L2s

LET d = RECEIVE Ms_dlg_in IN
    CASE d
        '0X : wait2();
        '10 : wait2();
        '11 : buff_to_L2s();
    ENDCASE
ENDLET
ENDROUTINE wait2
```

```
ROUTINE int_L1

! interrogate L1

int_count <- 0;
do_L1_map();
LET a = RECEIVE L1_status_in IN
    CASE a
         '0X : ! L2DLAT miss
               load_L2DLAT();
               do_L1_map();
         '11 : ! L1 hit
              LET b = RECEIVE L1DIR.DL IN
                  CASE b
                       '0 : invalid_L1DIR();
                       '1 : L1_to_out_buf(); ! not shown
                  ENDCASE ENDLET
    ENDCASE ENDLET
int_loop();
invalid_L2DLAT();
ENDROUTINE int_L1



ROUTINE int_loop        ! int_L1 main loop

do_L1_map();
int_count <- int_count + 1;
IF (over = '0) THEN int_loop();
ENDROUTINE int_loop



ROUTINE do_L1_map

! L1 returns status of mapping in L1_status_in

LET VA = RECEIVE virt_addr IN
    SEND VA(18:12) AT L2DLAT_addr;          ! index DLAT
    SEND (VA(12) || int_cnt) AT L1DIR_addr; ! index DIR
    SEND '10 AT L1_req_out;                 ! req L1 mapping
ENDLET
ENDROUTINE invalid_L2DLAT
```

```
ROUTINE invalid_L2DLAT

LET VA = RECEIVE virt_addr IN
    SEND VA(18:12) AT L2DLAT_addr;
    SEND '0 AT L2DLAT.VE;
    SEND '00 AT L1_req_out;
ENDLET
ENDROUTINE invalid_L2DLAT



ROUTINE load_L2DLAT

LET VA = RECEIVE virt_addr IN
    SEND VA(18:12) AT L2DLAT_addr;
    SEND rep_addr AT L2DLAT.TAG;
    SEND VA(43:23) AT L2DLAT.VK;
    SEND '1 AT L2DLAT.VE;
    SEND '00 AT L1_req_out;
ENDLET
ENDROUTINE load_L2DLAT



ROUTINE invalid_L1DIR

LET VA = RECEIVE virt_addr IN
    SEND (VA(12) || int_cnt) AT L1DIR_addr;
    SEND '0 AT L1DIR.VE;
    SEND '01 AT L1_req_out;
ENDLET
ENDROUTINE invalid_L1DIR



ROUTINE buff_to_L2s       ! transfer buffer to L2 store

count <- 0;
trnsfr_addr <- RECEIVE in_addr;
in_trnsfr();
ENDROUTINE buff_to_L2s



ROUTINE in_trnsfr         ! transfer loop for buff_to_L2s

in_reg<bindx> <- in_bread(cnt);
write(trnsfr_addr || cnt || '0000);
count <- count + 1;
IF (ovrflw = '0) THEN in_trnsfr();
ENDROUTINE in_trnsfr
```

```
ROUTINE L2s_to_buff

! Transfer L2s pg to buffer.  If any L2 dirty lines
! are written, set wb flag, indicating no buffer
! bypass on Ms transfer.

wb <- '0;                        ! initially no write-back
out_reg <- read_L2DIR(rep_addr);
WITH out_reg<sel,> DO
IF DL(0) = '1 THEN               ! for each dirty line,
    wb <- '1;                    ! L2 write-back
    count <- 0;                  ! load buffer sect
    out_trnsfr(0); ENDIF
IF DL(1) = '1 THEN
    wb <- '1;                    ! L2 write-back
    count <- 0;
    out_trnsfr(1); ENDIF
IF DL(2) = '1 THEN
    wb <- '1;                    ! L2 write-back
    count <- 0;
    out_trnsfr(2); ENDIF
IF DL(3) = '1 THEN
    wb <- '1;                    ! L2 write-back
    count <- 0;
    out_trnsfr(3); ENDIF
SEND '01 AT Ms_dlg_out;             ! signal buffer full to Ms
ENDROUTINE L2s_to_buff


ROUTINE out_trnsfr(lineno(1:0))

! transfer loop for L2s_to_buff
! only do transfer if L1 has not already
! written back that portion of out_buf

IF (OBV(lineno || count(5:3)) = '0) THEN
    read(rep_addr || lineno || cnt || '0000);
    out_ibreg <- out_reg;
    out_bwrite(lineno || cnt);
ENDIF
count <- count + 1;
IF (ovrflw = '0) THEN out_trnsfr(lineno);
ENDROUTINE out_trnsfr
```

```
ROUTINE do_L1_trnsfr

! L2 to L1 transfer : eight 16B lines

count <- 0;
L1_trnsfr_loop();
ENDROUTINE do_L1_trnsfr



ROUTINE L1_trnsfr_loop

LET d = read(map_addr || count(2:0) || '0000) IN
    SEND d AT line_out;
ENDLET
count <- count + 1;
IF (ovr = '0) THEN L1_trnsfr_loop();
ENDROUTINE L1_trnsfr_loop



ROUTINE do_L2_trnsfr

! L1 to L2 transfer : eight 16B lines

LET a = RECEIVE L1_addr IN
    out_reg <- read_L2DIR(a(14:5));
    DL(a(4:3)) <- '1;                ! mark 1KB line dirty
    in_reg <- out_reg;
    write(addr_reg);
    count <- 0;
    L2_trnsfr_loop();
ENDLET
ENDROUTINE do_L2_trnsfr



ROUTINE L2_trnsfr_loop

LET d = RECEIVE line_in,
    a = RECEIVE L1_addr IN
    in_reg <- d;
    write(a || count(2:0) || '0000);
    count <- count + 1;
    IF (ovr = '0) THEN L2_trnsfr_loop();
count <- count + 1;
IF (ovr = '0) THEN L1_trnsfr_loop();
ENDROUTINE L1_trnsfr_loop
```

```
SYSTEM L2r                           ! L2 replacement unit

CARRIERS      ! L2r

MEMORY L2rs<0:63>(63:0)              ! access count store

REGISTER in_reg<0:15>(3:0),          ! staging registers for L2rs
     AC IS in_reg(3:2),              ! two bit access count
     NR IS in_reg(1),                ! never replace flag
     DR IS in_reg(0);                ! don't replace flag

     addr_reg(6:0),                  ! address register for L2rs
     OVRFLW IS addr_reg(6);          ! overflow bit

     L2rQ<3:0>(9:0),                 ! replacement queue
     Qreg(4:0),                      ! search state vector
     Qhead(1:0),                     ! points to current best choice
     vQe(3:0);                       ! valid queue entry vector

FLIP-FLOP  Qempty,                   ! no valid queue entries
           wren,                     ! write enable
           rden;                     ! read enable


BEHAVIOR      ! L2r

main_rep_loop();


DEFINITIONS


ROUTINE main_rep_loop

LET r = RECEIVE req IN
    CASE r
        '0XX : update();
        '100 : reset();
        '101 : sendQe();
        '110 : sweep();
        '111 : search();
    ENDCASE
ENDLET
main_rep_loop();
ENROUTINE main_rep_loop
```

```
ROUTINE read_L2rs

! read from access count store

rden <- '1;
in_reg <- L2rs<addr_reg>;
ENDROUTINE read_L2rs


ROUTINE write_L2rs

! write into access count store

wren <- '1;
L2rs<addr_reg> <- in_reg;
ENDROUTINE  ! write_L2rs


ROUTINE sendQe

! send valid rplcmnt choice to requestor

IF Qempty THEN sweep();
               search();
               sendQe();
          ELSE SEND L2rQ<Qhead> AT L2LRU;
               checkQ(L2rQ<Qhead>);
ENDIF
ENDROUTINE sendQe


ROUTINE reset

! reset access count store

addr_reg <- 0;
reset_loop();
ENDROUTINE  ! reset


ROUTINE reset_loop

in_reg.AC <- '00;
in_reg.NR <- '0;
in_reg.DR <- '0;
write_L2rs();
addr_reg <- addr_reg + 1;
IF (OVRFLW = '0) THEN reset_loop(); ENDIF
ENDROUTINE reset_loop
```

```
ROUTINE update

! set access count entry accordingly

LET    a = RECEIVE addr
       r = RECEIVE req
    mask = RECEIVE addr(3:0) IN
    addr_reg <- a(9:4);
    read_L2rs();
    CASE r
        '000 : in_reg<mask>.AC <- '11;
               checkQ(a);
        '001 : in_reg<mask>.DR <- '1;
               checkQ(a);
        '010 : in_reg<mask>.DR <- '0;
        '011 : in_reg<mask>.NR <- '1;
               checkQ(a);
    ENDCASE
    write_L2rs;
ENDLET
ENDROUTINE update



ROUTINE checkQ(addr(9:0))

! invalidate referenced queue entry

IF (L2rQ<0> = addr) THEN vQe(0) <- '0; ENDIF
IF (L2rQ<1> = addr) THEN vQe(1) <- '0; ENDIF
IF (L2rQ<2> = addr) THEN vQe(2) <- '0; ENDIF
IF (L2rQ<3> = addr) THEN vQe(3) <- '0; ENDIF
IF (vQe(0) = '1) THEN Qhead <- '00;
                 ELSE
IF (vQe(1) = '1) THEN Qhead <- '01;
                 ELSE
IF (vQe(2) = '1) THEN Qhead <- '10;
                 ELSE
IF (vQe(3) = '1) THEN Qhead <- '11;
                 ELSE Qempty <- '1;
ENDIF ENDIF ENDIF ENDIF
ENDROUTINE checkQ



ROUTINE sweep

addr_reg <- 0;
sweep_loop();
ENDROUTINE sweep
```

```
ROUTINE sweep_loop

! process 16 entries in parallel

read_L2rs();
in_reg<0>.AC <-in_reg<0>.AC - 1; in_reg<1>.AC<-in_reg<1>.AC-1;
in_reg<2>.AC <-in_reg<2>.AC - 1; in_reg<3>.AC<-in_reg<3>.AC-1;
in_reg<4>.AC <-in_reg<4>.AC - 1; in_reg<5>.AC<-in_reg<5>.AC-1;
in_reg<6>.AC <-in_reg<6>.AC - 1; in_reg<7>.AC<-in_reg<7>.AC-1;
in_reg<8>.AC <-in_reg<8>.AC - 1; in_reg<9>.AC<-in_reg<9>.AC-1;
in_reg<10>.A <-in_reg<10>.A-1; in_reg<11>.AC<-in_reg<11>.AC-1;
in_reg<12>.AC<-in_reg<12>.AC-1;in_reg<13>.AC<-in_reg<13>.AC-1;
in_reg<14>.AC<-in_reg<14>.AC-1;in_reg<15>.AC<-in_reg<15>.AC-1;
write_L2rs();
addr_reg <- addr_reg + 1;
IF (OVRFLW = '0) THEN sweep_loop(); ENDIF
ENDROUTINE sweep_loop


ROUTINE search

addr_reg <- 0;
search_loop();
ENDROUTINE search


ROUTINE search_loop

read_L2rs();
IF get_zero() THEN put_inQ(); ENDIF
addr_reg <- addr_reg + 1;
IF (OVRFLW = '0) THEN search_loop();
ENDROUTINE search_loop


ROUTINE put_inQ

! simple queue update scheme
! more elaborate algorithm may be needed

IF (vQe(0) = '0) THEN put(0);
                 ELSE
IF (vQe(1) = '0) THEN put(1);
                 ELSE
IF (vQe(2) = '0) THEN put(2);
                 ELSE put(3);
ENDIF ENDIF ENDIF
ENDROUTINE    ! put_inQ
```

```
ROUTINE put(where)

vQe(where) <- '1;
L2rQ(where) <- (addr_reg || Qreg);
Qempty <- '0;
Qhead <- where;
ENDROUTINE   ! put


ROUTINE get_zero RETURNS BOOLEAN

! prioritizes zeros among 16 entries
! returns '0 if no zeros

IF (in_reg<0>.AC  = '00) THEN Qreg <- 0; RETURN '1; ENDIF
IF (in_reg<1>.AC  = '00) THEN Qreg <- 1; RETURN '1; ENDIF
IF (in_reg<2>.AC  = '00) THEN Qreg <- 2; RETURN '1; ENDIF
IF (in_reg<3>.AC  = '00) THEN Qreg <- 3; RETURN '1; ENDIF
IF (in_reg<4>.AC  = '00) THEN Qreg <- 4; RETURN '1; ENDIF
IF (in_reg<5>.AC  = '00) THEN Qreg <- 5; RETURN '1; ENDIF
IF (in_reg<6>.AC  = '00) THEN Qreg <- 6; RETURN '1; ENDIF
IF (in_reg<7>.AC  = '00) THEN Qreg <- 7; RETURN '1; ENDIF
IF (in_reg<8>.AC  = '00) THEN Qreg <- 8; RETURN '1; ENDIF
IF (in_reg<9>.AC  = '00) THEN Qreg <- 9; RETURN '1; ENDIF
IF (in_reg<10>.AC = '00) THEN Qreg <-10; RETURN '1; ENDIF
IF (in_reg<11>.AC = '00) THEN Qreg <-11; RETURN '1; ENDIF
IF (in_reg<12>.AC = '00) THEN Qreg <-12; RETURN '1; ENDIF
IF (in_reg<13>.AC = '00) THEN Qreg <-13; RETURN '1; ENDIF
IF (in_reg<14>.AC = '00) THEN Qreg <-14; RETURN '1; ENDIF
IF (in_reg<15>.AC = '00) THEN Qreg <-15; RETURN '1; ENDIF
RETURN '0;           ! no zeros
ENDROUTINE get_zero


ENDSYS    ! L2
```

Appendix C : File Manager Structures

The file manager data structures are summarized in this section. The description is in standard PASCAL, with the following notational modifications.

- "(.",".)" are used for square brackets.
- 2exp35 represents 2^{35}.
- @ is used for up-arrow (pointer definition).

```
CONST

virtual_space_size = 2exp44;
   virtual_pg_size = 2exp16;
          seg_size = 2exp29;
        pg_per_seg = seg_size DIV virtual_pg_size;
            seg_no = virtual_space_size DIV seg_size;


TYPE

          pg_rng = 0..pg_per_seg-1;
         SID_rng = 0..seg_no-1;
         TID_rng = 0..2exp10-1; /* assume 1024 TIDs   */
       lock_type = (X,SIX,IX,IS,S,N);
    segment_type = (seq,rand,temp,perm);
protect_key_type = (RO,RW,EO);  /* protection modes   */

                              /* 8KB BsA table entry   */
                              /* BsA_type is 8 bytes   */
BsAT_entry_type = ARRAY(.0..1023.) OF BsA_type;

                              /* link can hold 8 ptrs  */
                              /* or 2 actual BsAs      */
                              /*BsAT_ptr is 2 bytes    */
SDT_link = ARRAY (.0..7.) OF BsAT_ptr;
```

```
SDT_header = RECORD
   FPT : file_pt;            /* ptr to CLIST          */
   SL : pg_per_seg;          /* segment length        */
   ST : segment_type;        /* segment type          */
   CASE SEQ : ST OF          /* sequential file       */
     seq: ( PPM : pg_per_seg, /* pre-paging marker    */
            FOB : (F,R));    /* forward or reverse    */
     END;
   PK : protect_key_type;    /* protection keys       */
   FLV,                      /* CLIST ptr valid       */
   FF,                       /* free entry flag       */
   AS : BOOLEAN;             /* active segment flag   */
   FL,                       /* free list ptr if free */
   BSID : SID_rng;           /* SID of BsAT entry     */
   ASTI : AST_rng;           /* pointer to AST entry  */
                             /* active window:        */
   LP,                       /* last page no          */
   FP : page_rng;            /* first page no         */
   END;

file_entry = RECORD
   max_lock : lock_type;     /* maximum lock          */
   BFPT,                     /* backward file ptr     */
   FFPT : file_ptr;          /* forward file link ptr */
   GGPT,                     /* granted group ptr     */
   WGPT : group_ptr;         /* waiting group ptr     */
   GGV,                      /* valid granted group   */
   WGV,                      /* valid waiting group   */
   VPE,                      /* valid page entry      */
   BOC,                      /* beginning of chain    */
   EOC : BOOLEAN;            /* end of chain          */
   PPT : page_ptr;           /* ptr to page entry     */
   END;

page_entry = RECORD
   max_lock : lock_type;     /* maximum lock          */
   page_no : pg_rng;         /* relative page number  */
   FPPT : page_ptr;          /* forward page link ptr */
   GGPT,                     /* granted group ptr     */
   WGPT : group_ptr;         /* waiting group ptr     */
   GGV,                      /* valid granted group   */
   WGV,                      /* valid waiting group   */
   VLE,                      /* valid line entry      */
   BOC,                      /* beginning of chain    */
   EOC : BOOLEAN;            /* end of chain          */
   CASE x : BOC OF           /* backward file ptr     */
    1 : (BPPT : page_ptr);   /* points to either page */
    0 : (BPPT : file_ptr);   /* or to file entry      */
    END;
   LPT : line_ptr;           /* ptr to line entry     */
   END;
```

```
line_entry = RECORD
    max_lock : lock_type;    /* maximum lock           */
    line_no : 0..15;         /* relative line number   */
                             /* assuming 4KB lock line */
    FLPT : line_ptr;         /* forward line link ptr  */
    GGPT,                    /* granted group ptr      */
    WGPT : group_ptr;        /* waiting group ptr      */
    GGV,                     /* valid granted group    */
    WGV,                     /* valid waiting group    */
    BOC,                     /* beginning of chain     */
    EOC : BOOLEAN;           /* end of chain           */
    CASE x : BOC OF          /* backward file ptr      */
     1 : (BPPT : line_ptr);  /* points to either line  */
     0 : (BPPT : page_ptr);  /* or to page entry       */
     END;
    END;

group_entry = RECORD
    held_lock : lock_type;
    TID : TID_rng;             /* transaction id          */
    TPT,                       /* TID chain link ptr      */
    FGPT : group_ptr;          /* forward group link ptr  */
    BOC,                       /* beginning of chain      */
    EOC : BOOLEAN;             /* end of chain            */
    CASE x : BOC OF            /* backward group ptr      */
      1 : (BGPT : group_ptr);  /* points to either group  */
      0 : (BGPT : file_ptr);   /* or to file entry        */
     END;
    END;

file_ptr = @file_entry;
page_ptr = @page_entry;
line_ptr = @line_entry;
group_ptr = @group_entry;
```

```
VAR

                                    /* seg descriptor table    */
SDT : ARRAY (.0..seg_no-1.) OF RECORD
    header : SDT_header;
    link : SDT_link;
    END;

                                    /* backup store addr table */
BsAT : FILE OF BsAT_entry_type;

                                    /* single page indir. BsAT */
                                    /* one entry per BsAT page */
IBsAT : ARRAY (.0..8191.) OF BsA_type;

                               /* one flag for each BsAT entry */
BsATM : ARRAY (.0..2exp16-1.) OF BOOLEAN;

                                    /* 8 page active seg table */
                                    /* entry holds either BsA  */
                                    /* or hole descriptor      */
AST : ARRAY (.0..2exp16-1.) OF BsA_type;
```

References

(AGRA77) O. P. Agrawal and A. V. Pohm, "Cache Memory Systems for Multiprocessor Architecture," *Proceedings of the Spring Joint Computer Conference,* June 1977, 955-964.

(ALLE80) A. O. Allen, "Queueing Models of Computer Systems," *Computer,* vol. 13, no. 4, April 1980, 13-31.

(AMDA70) G. M. Amdahl, "Storage and IO Parameters and Systems Potential," *IEEE Computer Group Conference (COMCON),* 1970, 371-372.

(ARNO74) J. S. Arnold, D. P. Casey and R. H. McKinstry, "Design of Tightly-Coupled Multiprocessing Programming," *IBM System Journal,* vol. 13, no. 1, 1974, 60-87.

(BAER80) J. Baer, *Computer Systems Architecture,* Computer Science Press Inc., Rockville, 1980.

(BAIN81) W. L. Bain, Jr. and S. R. Ahuja, "Performance Analysis of High-Speed Digital Buses for Multiprocessing Systems," *Proceedings of the 8th IEEE Symposium on Computer Architecture,* May 1981, 107-133.

(BEAU78) M. D. Beaudry, "Performance-Related Reliability Measures for Computing Systems," *IEEE Transactions on Computers,* vol. C-27, no. 6, June 1978, 540-547.

(BELL74) J. Bell, D. Casasent and G. Bell, "An Investigation of Alternative Cache Organizations," *IEEE Transactions on Computers,* vol. C-23, no. 4, April 1974, 346-351.

(BENS72) A. Bensoussan, C. T. Clingen and R. C. Daley, "The MULTICS Virtual Memory: Concepts and Design," *Communications of the ACM,* vol. 15, no. 5, May 1972, 308-318.

(BERG79) N. C. Berglund, "Level-Sensitive Scan Design Tests Chips, Boards, System," *Electronics,* March 15, 1979, 108-110.

(BRAD80) J. T. Brady, J. D. Calvert, C. V. Freiman and R. T. Schmalz, "Integrated Paging Storage," *IBM Technical Disclosure Bulletin,* vol. 23, no. 7A, December 1980, 2908-2909.

(BRAN81a) A. Brandwajn, "A Capacity Planning Model of a DASD Subsystem," *Performance '81,* F. J. Kylstra, ed., North-Holland Publishing Co., Amsterdam,1981, 401-413.

(BRAN81b) A. Brandwajn, "Multiple Paths Versus Memory for Improving DASD Subsystem Performance," *Performance '81,* F. J. Kylstra, ed., North-Holland Publishing Co., Amsterdam,1981, 415-434.

(BURS80) D. Bursky, "PROMs and RAMs Remember in Many Ways and Styles," *Electronic Design,* July 1980, 86-92.

(BURS81a) D. Bursky, "1981 Technology Forecast: Digital LSI," *Electronic Design,* January 1981, 85-122.

(BURS81b) D. Bursky, "Semiconductor Special: Digital," *Electronic Design,* June 1981, 92-104.

(CALE78) S. Calebotta, "System Design with Dynamic Memories Takes Attention to Detail," *Electronics,* February 1978, 109-113.

(CHAM73) D. D. Chamberlin, S. H. Fuller and L. Y. Liu, "An Analysis of Page Allocation Strategies for Multiprogramming Systems with Virtual Memory," *IBM Journal of Research and Development,* vol. 17, April 1973, 404-412.

(CHAM81) D. D. Chamberlin et al., "A History and Evaluation of System R," *Communications of the ACM,* vol. 24, no. 10, October 1981, 632-646.

(CHAN80) S. T. Chanson and P. S. Sinha, "Optimization of Memory Hierarchies in Multiprogrammed Computer Systems With Fixed Cost Constraint," *IEEE Transactions on Computers,* vol. C-29., no. 7., July 1980., 611-618.

(CHOW74) C. K. Chow, "On Optimization of Storage Hierarchies," *IBM Journal of Research and Development,* vol. 18, May 1974, 194-203.

(CHOW76) C. K. Chow, "Determination of Cache's Capacity and its Matching Storage Hierarchy," *IEEE Transactions on Computers,* vol. C-25, no. 2, February 1976, 157-164.

(COCK81) J. Cocke, personal communications.

(COMP78) "M/600 Doubles DG Mini Power," *ComputerWorld,* January 30, 1978, 1,8.

(COMP79) "Disk Cache System Added For Memorex 3650 Drives," *ComputerWorld,* July 2, 1979, 29.

(DALE68) R. C. Daley and J. B. Dennis, "Virtual Memory, Processes and Sharing in MULTICS," *Communications of the ACM,* vol. 11, no. 5, May 1968, 306-312.

(DENG80) P. J. Denning, "Working Sets Past and Present," *IEEE Transactions on Software Engineering,* vol. SE-6, no. 1, January 1980, 64-84.

(DENN65) J. B. Dennis, "Segmentation and the Design of Multiprogrammed Computer Systems," *Journal of the ACM,* vol. 12, no. 4, October 1965, 589-602.

(DENN81) J. B. Dennis, "HEX - Hardware EXperimental Description Notation," *Laboratory for Computer Science MIT,* Cambridge, February 1981.

(DEPE76) R. H. Dependahl, Jr. and L. Presser, "File Input/Output Control Logic," *Computer,* vol. 9, no. 10, October 1976, 38-42.

(DESA80) S. M. Desai, "System Cache for High Performance Processors," *IBM Technical Disclosure Bulletin,* vol. 23, no. 7A, December 1980, 2915-2917.

(DORA76) R. W. Doran, "Virtual Memory," *Computer,* vol. 9, no. 10, October 1976, 27-37.

(DRAK67) A. W. Drake, *Fundamentals of Applied Probability Theory,* McGraw-Hill Inc., New York, 1967.

(EAST79) M. C. Easton and P. A. Franaszek, "Use Bit Scanning in Replacement Decisions," *IEEE Transactions on Computers,* vol. C-28, no. 2, February 1979, 133-141.

(FEIE71) R. J. Feiertag and E. I. Organick, "The MULTICS Input/Output System," *Proceedings of the 3rd Symposium on Operating Systems Principles.* October 1971, 35-41.

(GARD80) G. Gardarin, "Integrity, Consistency, Concurrency, Reliability in Distributed Database Management Systems," *Distributed Data Bases,* Bracchi/Nijssen, eds., North-Holland Publishing Co., Amsterdam, 1980, 335-351.

(GECS74) J. Gecsei and J. A. Lukes, "A Model for the Evaluation of Storage Hierarchies," *IBM Systems Journal,* vol. 13, no. 2, 1974, 163-178.

(GEIS82) R. Geist and K. Trivedi, "Queueing Network Models in Computer System Design," *Math Magazine,* no. 15, March 1982.

(GOTO77) E. Goto, T. Ida and T. Gunji, "Parallel Hashing Algorithms," *Information Processing Letters,* vol. 6, no. 1, February 1977, 8-13.

(GRAY78) J. Gray, "Notes on Data Base Operating Systems," *IBM Research Report (RJ-2188),* San Jose, February 1978.

(GUPT78) R. K. Gupta and M. A. Franklin, "Working Set and Page Fault Frequency Paging Algorithms: A Performance Comparison," *IEEE Transactions on Computers,* vol. C-27, no. 8, August 1978, 706-712.

(HABE76) A. N. Habermann, *Introduction to Operating System Design,* Science Research Associates Inc., Chicago, 1976.

(HARD79) T. Harder and A. Reuter, "Optimization of Logging and Recovery in a Database System," *Data Base Architecture,* Bracchi/Nijssen, eds., North-Holland Publishing Co., Amsterdam, June 1979, 151-168.

(HART81) R. Hartmann and R. Walker, "LSI Gate Arrays Outpace Standard Logic," *Electronic Design,* March 1981, 107-112.

(HAUG75) K. E. Haughton, "An Overview of Disk Storage Systems," *Proceedings of the IEEE,* vol. 63, no. 8, August 1975, 1148-1152.

(HEID81) P. Heidelberger and K. S. Trivedi, "Queueing Network Models for Parallel Processing with Asynchronous Tasks," *IBM Research Report (RC-9102),* IBM Corp., October 1981.

(HIVE78) J. Hively, "Subnanosecond ECL Gate Array," *Fairchild Journal of Semiconductor Progress,* vol. 6, no. 3, June 1978, 3-9.

(HORO77) E. Horowitz and S. Sahni, *Fundamentals of Data Structures,* Computer Science Press, Inc., Potomac, 1977.

(HOUD79) M. E. Houdek and G. R. Mitchell, "Hash Index Helps Manage Large Virtual Memory," *Electronics,* March 15, 1979, 111-113.

(HOUD81) M. E. Houdek, F. G. Soltis and R. L. Hoffman, "IBM System/38 Support for Capability-Based Addressing," *Proceedings of the 8th IEEE Symposium on Computer Architecture,* May 1981, 341-348.

(IBM81) IBM Corp., *IBM System/370 Principles of Operation (GA22-7000-8),* October 1981.

(JONE81a) D. A. Jones, "Disk Workload Characterization From Event Trace Analyses," *Performance '81,* F. J. Kylstra, ed., North-Holland Publishing Co., Amsterdam, 1981, 301-313.

(JONE81b) D. A. Jones, "Statistical Investigation of Factors Affecting Filestore Performance," *Performance '81,* F. J. Kylstra, ed., North-Holland Publishing Co., Amsterdam, 1981, 315-333.

(KAPL73) K. R. Kaplan and R. O. Winder, "Cache-based Computer Systems," *Computer,* vol. 6, no. 3, March 1973, 30-36.

(KATZ71) H. Katzan Jr., "Storage Hierarchy Systems," *Proceedings of the Spring Joint Computer Conference (AFIPS),* vol. 38, 1971, 325-336.

(KING81) P. J. King and I. Mitrani, "The Effect of Breakdown on the Performance of Multiprocessor Systems," *Performance '81,* F. J. Kylstra, ed., North-Holland Publishing Co., Amsterdam, 1981, 201-211.

(KNUT73) D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching,* Addison-Wesley Publishing Co. Inc., Reading, 1973.

(LANG76) G. G. Langdon, Jr. "A Figure-of-Merit Approach to a Two Level Hierarchy for a Disk," *IBM Research Report,* RJ-1778, San Jose, May 1976.

(LANG77) T. Lang, C. Wood and E. B. Fernandez, "Database Buffer Paging in Virtual Storage Systems," *ACM Transactions on Database Systems,* vol. 2, no. 4, December 1977, 339-351.

(LIN72) Y. S. Lin and R. L. Mattson, "Cost-Performance Evaluation of Memory Hierachies," *IEEE Transactions on Magnetics,* vol. MAG-8, September 1972, 390-392.

(LIPT68) J. S. Liptay, "Structural Aspects of the System/360 Model 85: II The Cache," *IBM Systems Journal,* vol. 7, no. 1, 1968, 15-21.

(LIST75) A. M. Lister, *Fundamentals of Operating Systems,* Macmillan Press Ltd., London 1975.

(MACD75) J. E. MacDonald and K. L. Sigworth, "Storage Hierarchy Optimization Procedure," *IBM Journal of Research and Development,* vol. 19, March 1975, 133-140.

(MACK74) R. A. MacKinnon, "Advanced Function Extended with Tightly-Coupled Multiprocessing," *IBM System Journal,* vol. 13, no. 1, 1974, 32-59.

(MARU75a) K. Maruyama, "mLRU Page Replacement Algorithm in Term of the Reference Matrix," *IBM Technical Disclosure Bulletin,* vol. 17, no. 10, March 1975, 3101-3103.

(MARU75b) K. Maruyama, "mLRU Page Replacement Algorithm in Term of the Reference Counters," *IBM Technical Disclosure Bulletin,* vol. 17, no. 10, March 1975, 3104-3106.

(MATT70) R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal,* vol. 9, no. 2, 1970, 78-117.

(MATT71) R. L. Mattson, "Evaluation of Multilevel Memories," *IEEE Transactions on Magnetics,* vol. MAG-7, December 1971, 814-819.

(MAY81) C. M. May, "Management Technique for Memory Hierarchies," *IBM Technical Disclosure Bulletin,* vol. 24, no. 1A, June 1981, 333-335.

(MEAD70) R. M. Meade, "On Memory System Design," *Proceedings of the Fall Joint Computer Conference (AFIPS),* vol. 37, 1970, 33-43.

(NAUM79) J. S. Nauman, "Observations on Sharing in Data Base Systems," *Data Base Architecture,* Bracchi/Nijssen, eds., North-Holland Publishing Co., Amsterdam, June 1979, 75-94.

(POHM73) A. V. Pohm, "An Efficient Flexible Buffered Memory System," *IEEE Transactions on Magnetics,* vol. MAG-9, September 1973, 173-179.

(POHM75a) A. V. Pohm, et. al, "Cost/Performance Perspectives of Paging with Electronic and Electromechanical Backing Stores," *Proceedings of the IEEE,* vol. 63, no. 8, August 1975, 1123-1128.

(POHM75b) A. V. Pohm, O. P. Agrawal and R. N. Monroe, "The Cost and Performance Tradeoffs of Buffered Memories," *Proceedings of the IEEE,* vol. 63, no. 8, August 1975, 1129-1135.

(POHM81) A. V. Pohm and T. A. Smay, "Computer Memory Systems," *Computer,* vol. 14, no. 10, October 1981, 93-110.

(POSA80a) J. G. Posa, "Dynamic RAMs: What to Expect Next," *Electronics,* May 1980, 119-129.

(POSA80b) J. G. Posa, "Gate Arrays," *Electronics,* September 1980, 145-158.

(POTI80) D. Porier and P. Leblanc, "Analysis of Locking Policies in Database Management Systems," *Communications of the ACM,* vol. 23, no. 10, October 1980, 584-593.

(POWE77) M. L. Powell, "The DEMOS File System," *Proceedings of the 6th ACM Symposium on Operating Systems Principles,* November 1977, 33-42.

(PRIO) J. Prioste, *MECL 10,000 Macrocell Array Design Manual (preliminary),* Motorola Corp., Phoenix, Arizona.

(PUGH71) E. W. Pugh, "Storage Hierarchies: Gaps, Cliffs and Trends," *IEEE Transactions on Magnetics,* vol. MAG-7, December 1971, 810-814.

(QUIT81) P. Quittner, etc., "Comparison of Synonym Handling and Bucket Organization Methods," *Communications of the ACM,* vol. 24, no. 9, September 1981, 579-583.

(RAMA70) C. V. Ramamoorthy and K. M. Chandy, "Optimization of Memory Hierarchies in Multiprogrammed Systems," *Journal of the ACM,* vol. 17, no. 3, July 1970, 426-445.

(RAO78) G. S. Rao, "Performance Analysis of Cache Memories," *Journal of the ACM,* vol. 25, no. 3, July 1978, 378-395.

(REGE76) S. L. Rege, "Cost, Performance and Size Tradeoffs for Different Levels in a Memory Hierarchy," *Computer,* vol. 9, April 1976, 43-51.

(RIES77) D. R. Ries and M. R. Stonebraker, "Effects of Locking Granularity in a Database Management System," *ACM Transactions on Database Systems,* vol. 2, no. 3, September 1977, 233-246.

(RIES79) D. R. Ries and M. R. Stonebraker, "Locking Granularity Revisited," *ACM Transactions on Database Systems,* vol. 4, no. 2, June 1979, 210-227.

(SALT74) J. H. Saltzer, "A Simple Linear Model of Demand Paging Performance," *Communications of the ACM,* vol. 17, no. 4, April 1974, 181-186.

(SAUE77) C. H. Sauer, M. Reiser and E. A. MacNair, "RESQ - A Package for Solution of Generalized Queueing Networks," *Proceedings of the Joint Computer Conference (AFIPS),* vol. 46, 1977, 977-986.

(SAUE80a) C. H. Sauer, E. A. MacNair and S. Salza, "A Language for Extended Queuing Network Models," *IBM Journal of Research and Development,* vol. 24, no. 6, November 1980.

(SAUE80b) C. H. Sauer and K. M. Chandy, "Approximate Solution of Queueing Models," *Computer,* vol. 13, no. 4, April 1980, 25-32.

(SAUE81) C. H. Sauer and K. M. Chandy, *Computer Systems Performance Modeling,* Prentice Hall Inc., Englewood Cliffs, 1981.

(SCHN76) P. Schneider, "Working Set Restoration - A Method to Increase the Performance of Multilevel Storage Hierarchies," *Proceedings of the Joint Computer Conference (AFIPS),* vol. 45, 1976, 373-380.

(SHAW74) A. C. Shaw, *The Logical Design of Operating Systems,* Prentice-Hall Inc., Englewood Cliffs, 1974.

(SITE78) R. L. Sites, "An Analysis of the Cray-1 Computer," *Proceedings of the 5th IEEE Symposium on Computer Architecture,* April 1978, 101-106.

(SMIT76) A. J. Smith, "A Modified Working Set Paging Algorithm," *IEEE Transactions on Computers,* vol. C-25, no. 9, September 1976, 907-914.

(SMIT77) A. J. Smith, "Multiprocessor Memory Organization and Memory Interference," *Communications of the ACM,* vol. 20, no. 10, October 1977, 754-761.

(SMIT78a) A. J. Smith, "Directions for Memory Hierarchies and Their Components: Research and Development," *Proceedings of the IEEE Computer Software and Applications Conference (COMPSAC),* November 1978, 704-709.

(SMIT78b) A. J. Smith, "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory," *IEEE Transactions on Software Engineering,* vol. SE-4, no. 2, March 1978, 121-130.

(SMIT78c) A. J. Smith, "On the Effectiveness of Buffered and Multiple Arm Disks," *Proceedings of the 5th IEEE Symposium on Computer Architecture,* April 1978, 242-248.

(SMIT79) A. J. Smith, "Characterizing the Storage Process and Its Effect on the Update of Main Memory by Write Through," *Journal of the ACM,* vol. 26, no. 1, January 1979, 6-27.

(SPAR78) F. J. Sparacio, "Data Processing System with Second Level Cache," *IBM Technical Disclosure Bulletin,* vol. 21, no. 6, November 1978, 2468-2469.

(TANG76) C. K. Tang, "Cache System Design in the Tightly Coupled Multiprocessor System," *Proceedings of the Joint Computer Conference (AFIPS),* vol. 45, 1976, 749-753.

(TI80) Texas Instruments Inc., "TAT Series STL Logic Arrays," Dallas, 1980.

(TOKU80) T. Tokunaga, Y. Hirai and S. Yamamoto, "Integrated Disk Cache System with File Adaptive Control," *IEEE Computer Group Conference (COMCON),* 1980, 412-416.

(TRIV80) K. S. Trividi and R. E. Kinicki, "A Model for Computer Configuration Design," *Computer,* vol. 13, no. 4, April 1980, 47-54.

(TRIV81) K. S. Trivedi and T. M. Sigmon, "Optimal Design of Linear Storage Hierarchies," *Journal of the ACM,* vol. 28, no. 2, April 1981, 270-288.

(WELC78) T. A. Welch, "Memory Hierarchy Configuration Analysis," *IEEE Transactions on Computers,* vol. C-27, no. 5, May 1978, 408-413.

(YEN81) W. C. Yen and K. S. Fu, "Analysis of Multiprocessor Cache Organizations with Alternative Main Memory Update Policies," *Proceedings of the 8th IEEE Symposium on Computer Architecture,* May 1981, 89-105.

Biographical Note

Evan M. Tick was born in Flushing, NY. He received both his BS and MS degrees in Electrical Engineering from the Massachussetts Institute of Technology, Cambridge in 1982. Evan currently works on high-end machine organization for IBM at the T. J. Watson Research Center in Yorktown Heights, NY. He is a member of Eta Kappa Nu and Tau Beta Pi.