

Portland State University

PDXScholar

Dissertations and Theses

Dissertations and Theses

6-9-1993

Performance Analysis of a Hierarchical, Cache-Coherent, Shared Memory Based, Multi-processor System

Raman Nayyar

Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds



Part of the [Electrical and Computer Engineering Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Nayyar, Raman, "Performance Analysis of a Hierarchical, Cache-Coherent, Shared Memory Based, Multi-processor System" (1993). *Dissertations and Theses*. Paper 4695.

<https://doi.org/10.15760/etd.6579>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

AN ABSTRACT OF THE THESIS OF Raman Nayyar for the Master of Science in Electrical and Computer Engineering presented on June 09, 1993.

Title: Performance Analysis of a Hierarchical, Cache-Coherent, Shared Memory Based, Multi-processor System.

APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:



Michael A. Driscoll, Chair



Andrew M. Fraser



Thomas Schubert

We have conducted a performance analysis of a large scale multiprocessor system based on shared buses organized in a hierarchical fashion and employing an easy to implement snoopy cache protocol.

This arrangement, named TREEBUS [5], presents a logical extension path for multiprocessor systems based on a single shared bus whose scalability is limited by the available system bus bandwidth [26]. The multiple, independent, hierarchical buses overcome the bus bandwidth limitation and the architecture can scale to relatively large sizes.

We have developed an easy to use, reasonably accurate and computationally efficient analytic model for analyzing the performance of the memory hierarchy. Our analysis presents a balanced view by incorporating cost and size of the memory subsystem, two parameters which can significantly impact the feasibility of this architecture.

The results indicate that the TREEBUS can deliver high performance for a maximum of about 512 processors using available technology. For larger sizes, the problem is not the limited system bus bandwidth but the unmanageable size of the main memory and a deteriorating cost/performance ratio.

PERFORMANCE ANALYSIS OF A HIERARCHICAL, CACHE-COHERENT,
SHARED MEMORY BASED, MULTI-PROCESSOR SYSTEM

by

RAMAN NAYYAR

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1993

TO THE OFFICE OF THE GRADUATE STUDIES:

The members of the committee approve the thesis of Raman Nayyar presented on
June 09, 1993.



Michael A. Driscoll, Chairman

Andrew M. Fraser

Thomas Schubert

APPROVED:



Rolf Schaumann, Chair, Department of Electrical Engineering

Roy W. Koch, Vice Provost for Graduate Studies and Research

TABLE OF CONTENTS

	PAGE
LIST OF TABLES	vii
LIST OF FIGURES	viii
CHAPTER	
I INTRODUCTION	1
Thesis Organization	4
II NEED FOR MULTI-LEVEL CACHES IN A MULTIPROCESSOR SYSTEM	6
Multilevel Caches And System Performance	6
Match the processor and main memory speed	7
Match the processor and system bus bandwidth ...	9
New buses with increased bandwidth	10
Reduce the heavy duty traffic on the system bus	12
Reduce Overall System Cost	14
Importance of simulation in reducing overall system cost	16
Cache Coherence In Multiprocessor Systems	20
Important issues concerning coherence protocols ...	21
Cache Coherence Solutions In Multiprocessor Systems	22
Hardware-based protocols	23
Software-based protocols	27

	Multilevel Caches And Their Impact on Cache Coherence	29
	Multilevel Inclusion Principle (MLI)	30
	Characteristics and Limitations of Shared Bus Systems	32
	Evolution towards multi-level caches and multiple bus based microprocessor systems	32
III	TREEBUS ARCHITECTURE AND CACHE COHERENCE PROTOCOL	34
	TREEBUS Architecture	35
	Some definitions as applied to the TREEBUS architecture	37
	Cache coherence	39
	Data movement in the hierarchy -- an overview . . .	43
	Cache Directory Organization	45
	States of Cached blocks	45
	Implementation of Cache directory	51
	Cache Coherence Protocol (in detail)	53
	State transitions	53
	Flow charts for the coherence protocol	59
	Summary	75
IV	MODEL DEVELOPMENT	76
	Overview	76
	Model for Read accesses	76
	Functions in Read.c	77
	Variables used in Read.c	78
	Timing equations	79

Model for Write accesses	83
Functions in Write.c	83
Variables used	83
Timing equations for P_WRITE(1)	84
Cost Model	87
Mapping of High Level Parameters to Low Level Parameters	89
Mapping	90
V RESULTS AND FINAL ANALYSIS	92
Overview	92
Performance analysis	93
Send_data(i) and bus_access_time(i)	94
Supply_requesting_cache(i) and Update_parent_cache	94
Effect of Hit_cache[1] and Hit_peer[1]	95
Effect of Hit_cache[1] and Hit_cache[2]	97
Effect of P_clean[1] and P_clean[2] on P_write(1)	98
Effect of Peer_consistent[2] and Peer_consistent[3]	101
Effect of bus_access_time[1] and bus_access_time[2]	104
Effect of hit_cache[1] on processor's bandwidth requirements	108
Effect of MLI factor (α) on total memory size, cost and Cost/Performance	109
VI CONCLUSIONS AND FUTURE WORK	117
Conclusions	117
Maximize hit_cache[1]	117
Maximize localization of data sharing	118
Cost of the memory hierarchy is a limiting factor ..	119
TREEBUS still holds promise	119

VI	CONCLUSIONS AND FUTURE WORK	117
	Conclusions	117
	Maximize hit_cache[1]	117
	Maximize localization of data sharing	118
	Cost of the memory hierarchy is a limiting factor .	119
	TREEBUS still holds promise	119
	Validation	120
	Future Options for the Designer	121
	Develop a model using principles of queuing theory	121
	Simulate the model	121
	Build a prototype	122
	Future Work	122
	Incorporate memory contention	122
	Model Bus access time accurately	123
	Summary	124
	REFERENCES CITED	126
	APPENDICES	
	A READ.C	129
	B WRITE.C	135
	C HEADER FILES	138

LIST OF TABLES

TABLES		PAGE
I	Bandwidth of Different High Performance Buses	11
II	Characteristics of TREEBUS Architecture in Figure 1	39
III	Size of Individual Caches at each level	41
IV	Attributes Associated with Each State	49
V	Input Values to the Model: Set I	93
VI	Input Values to the Model: Set II (in bus cycles)	95
VII	Topology for Graphs in Figures 24-25	111

LIST OF FIGURES

FIGURE	PAGE
1. The TREEBUS architecture	35
2. State diagram for a cached block at level i.	54
3. State diagram for a cached block at level 1.	55
4. Read request at level 1.	59
5. Bus read request at the level i bus	60
6. Receive data from peer cache at level i	62
7. Supply data to the requesting cache	63
8. Recall(i)	65
9. Block replacement at level i	66
10. Send data to the level below	68
11. Overview of a P_write request	69
12. Invalidate process at level i	71
13. Sending Invalid acknowledge signal to level below	73
14. Invalidating blocks in descendant caches	74
15. Effect of hit_cache[1] and hit_peer[2] on average access time	96
16. Effect of hit_cache[1] and hit_cache[2] on average access time	98

17.	Read/Write access contribution to average access time, for various hit_cache[1] and hit_cache[2].	99
18.	Effect of p_clean[1] and p_clean[2] on average access time	100
19.	Effect of peer_consistent[2] and peer_consistent[3] on Recall(3).	102
20.	Effect of peer_consistent[2] and peer_consistent[3] on average access time	103
21.	Effect of bus_access_time[1] and bus_access_time[2] on average access time	105
22.	Effect of hit_cache[1] and bus_access_time[1] on average access time	106
23.	The TREEBUS architecture	107
24.	Effect of hit_cache[1] on bus bandwidth requirements	109
25.	MLI factor versus Total memory size.	112
26.	Effect of MLI factor on Cost ratio between unilevel and multilevel memory hierarchy	113
27.	Effect of MLI factor on Cost/Performance ratio comparison between unilevel and multilevel memory hierarchy.	114
28.	Cost/Performance ratio comparison between TREEBUS and an architecture similar to Sequent's Symmetry.	116

CHAPTER I

INTRODUCTION

Uniprocessor computer systems have reached a point where the performance gains with continuing technological advancements are only incremental and not enough to solve the large sized computing tasks, commonly referred to as the *grand challenges of computing* [31].

One of the most popular alternatives to substantially increase performance is by distributing tasks among a group of processors and allowing them to work in parallel. Multiprocessor systems based on the MIMD architecture (systems working on multiple instruction and multiple data streams at the same time) are very popular because "a system composed of ten one-MIPS processors is a much less formidable engineering effort than a ten-MIPS uniprocessor and its associated memory system." [29]

Systems based on MIMD type architecture are designed using shared or distributed memory. The salient features of the two types of systems are as follows:

Distributed memory: As the name suggests, the memory is distributed among the processors. A task involving data sharing necessitates sending messages to the processors involved in data sharing. The communication time (overhead) can easily dwarf the execution time. Also, the time taken to send a message to different processors is not the same. The further the processors are from each other, the higher the communication time becomes. The performance gain is realized by using a large

number of processors (as much as 1000) and partitioning the data so as to minimize the communication overhead.

A strong point in favor of this architecture is that it can be scaled to very large sizes as the size of the problem increases.

Shared memory: All the processors share the same main memory, thereby eliminating the need of sending messages between the different processors. There is a substantial saving in the communication overhead as compared to the distributed memory case, which makes this an attractive alternative.

An obvious problem because of data sharing is data access synchronization, i.e., while one processor is writing to a shared piece of data, another processor that wants to access it needs to wait until the write is completed. Also, since each processor has its local memory (cache), we have to ensure coherency of data across the system.

The simplest example of a shared memory multiprocessor system is multiple processors connected to a single common bus [26]. This system has limited scalability due to the system bus. Several alternatives have been proposed in the research literature to address this problem. A few of them are discussed in [2], [5], [6], [13], [19] [30], etc.

A problem common to all large scale multiprocessor based systems is the difficulty in analysis. There is limited or no history information available and traces from one architecture are probably useless for another. Also, there is not a clear understanding of the memory access patterns of parallel programs and besides, the

architecture itself influences the memory access pattern. Hence, a model proves to be a valuable analytical tool to give a first cut estimate to the designer.

In our thesis, we have focussed on a shared memory based architecture, in particular, the architecture discussed in [5]. The architecture is designed using a hierarchy of buses, called TREEBUS. The cache coherent multiprocessors use a hardware based cache coherence protocol.

In our opinion, the analysis in the original research for the TREEBUS architecture presents an over optimistic picture and projects it as a very high performance {commercially and practically feasible} system. We differ from this viewpoint and our analysis presents a balanced and realistic scenario by considering not only performance, but also practical (total memory size) and commercial feasibility (cost of the memory hierarchy) aspects.

We have analyzed the operation of the multilevel cache hierarchy and the cache coherence protocol in great detail and developed an analytical model which gives a good first cut estimate about the cost and performance aspects. The model is based on low level system parameters, e.g., hit/miss ratios, the probability of cached data blocks being in a particular state, etc. It is easy to use, reasonably accurate and computationally very efficient. It was validated by analyzing a single common bus based multiprocessor system and the results matched our expectations. The effect of all possible low level parameters on the performance of the TREEBUS hierarchy were studied.

The model will enable the designer to have a tool which will help him/her understand the worst case impact of different high level characteristics. For example,

if there is a high degree of data sharing (a high level characteristic), how is the average access time affected? Which low level parameter is most critical in this particular situation?

It is easy to figure out how all low level parameters will be affected by a particular high level characteristic, but the tough part is to come up with the exact nature of interdependencies among them. Using our model, the designer can bypass this complicated step and instead sweep the value of all the affected parameters over a wide range. Each iteration of the analysis can give the effect of two parameters on a certain output parameter, e.g., average access time. This range could even include values that normally would defy logic and intuition.

By following this approach, the designer can have an in depth understanding of the system response to a particular type of program behavior even without knowing the exact relationships between the different parameters. Each iteration of the model takes a few seconds of CPU time.

THESIS ORGANIZATION

The rest of the thesis is organized as follows: Chapter 2 discusses the need for multilevel caches in multiprocessor computer systems. The issues discussed are mismatch between processor speed and bandwidth to that of main memory speed and system bus bandwidth and reduction of overall system cost. We also discuss briefly the important issues in solving the cache coherence problem and the various efforts

presented in the research literature to solve it. Finally, we introduce our architecture of choice for this thesis.

Chapter 3 discusses the TREEBUS architecture and the cache coherence protocol in great detail. Chapter 3 lays the groundwork for the development of the mathematical model. The analytical model is developed in Chapter 4. We have focussed on performance as well as the cost aspect. A simple cost model is developed to help us analyze the cost/performance aspects. Chapter 5 covers the detailed analysis. The performance measured by average access time is affected by many parameters. At times, we also need to consider the impact of the same parameter at different levels in the system.

Chapter 6 covers the key conclusions from our study, recommendations, possible future work that can be done and a brief summary.

CHAPTER II

NEED FOR MULTI-LEVEL CACHES IN A MULTIPROCESSOR SYSTEM

The goal of every computer memory system designer is probably the same, i.e., *Achieve maximum throughput at minimum possible cost.* Caches help us meet this goal by substantially enhancing the performance while keeping the cost to a reasonable level. This chapter justifies the claim and addresses the various issues involved in designing a multilevel cache hierarchy for a multiprocessor system.

The chapter is divided into four sections. The first one justifies the need for a multi-level hierarchy and examines how the cache hierarchy improves the performance. The second section addresses the second part of the goal, i.e., cost reduction. The first two sections address the above mentioned goal directly. The third section discusses the architectural implications (potential problems) of multilevel caches and the various alternatives available to take care of the problems. There is a definite gain due to the caches, but there is also a cost to pay. After discussing caches in detail, the fourth section justifies our selection of a multilevel, hierarchical, cache-coherent multiprocessor system as the architecture of choice for detailed study.

MULTILEVEL CACHES AND SYSTEM PERFORMANCE

Ideally, if *all* the requests of the processor can be satisfied in one cycle, it would never have to wait for instructions or data. The processor in this case would be

performing at its theoretical best. In real life however, a vast disparity exists between the processor speed and that of main memory and the system bus. The following sub-sections explain the problem in detail and suggest a solution.

Match the processor and main memory speed

The need for caches is very closely tied with the rapid advancement in the field of microprocessor design over the last decade and a half and the slow pace of improvement in performance of system buses and main memory chips.

The fastest DRAMS available as of late 1992 had cycle times of around 50 ns [21] where as a 50-Mhz processor has a cycle time of only 20 ns. In other words, if a 50-Mhz processor was directly connected to a 50 ns DRAM, it would run with an efficiency of 33.33%. This is because it would spend 30 ns just waiting for the data to be made available from the main memory (2 wait states, if the Ready signal is sampled at the end of the clock period). Similarly, a 100 Mhz processor will have a cycle time of 10 ns, and hence an efficiency of only 20% (4 wait states are encountered).

Primary cache. A solution to the above problem is to have a primary (on-chip) cache that can match the CPU speed. The cycle time of an on-chip cache is less than that of the CPU. It can be a unified entity for both instruction and data or there can be separate caches for instructions and data. The size of on-chip caches available on present day microprocessors range from 4 Kbytes to 32 Kbytes.

It is not enough to just match the speed of the processor and cache, we must also ensure that all the processor's requests are satisfied in its local cache (hits). In this ideal situation, the requests would never need to go to the main memory via the slow

system bus for instructions or data. This would enable the processor to keep performing at its peak.

In other words, the hit-ratio of the primary cache should be as high as possible. If the hit ratio is low, the requests from the processor would need to go to the main memory frequently, thereby reducing the positive impact of the primary cache. Since the hit ratio is an increasing function of cache size, one is tempted to make it as large as can be accommodated on the die. The primary cache design at this point is a trade-off between space available on the die and the performance desired, e.g., the hit-ratio for the primary cache in Intel's 80486 processor is between 95-99% for DOS programs and 92-94% for Unix programs [9].

Secondary cache. While we try to increase the size of the cache to the maximum extent possible to achieve a high hit ratio, the access time of the cache also keeps on increasing. The size of the cache can only be increased to a certain point for performance reasons. Beyond this point, the increase in cache size adversely affects the speed of the cache (access time). This, in turn, degrades the overall system performance.

The cache designer resolves the trade-off issue of cache size and performance by having a multi-level cache hierarchy in the system. The primary cache is kept just large enough to match the CPU speed and provide a reasonable hit rate. A second level cache, called the secondary cache or the off-chip cache, is connected to the primary cache. The secondary cache is local to the CPU and is added to improve the overall hit ratio and system performance.

The secondary cache is made large enough to increase the overall hit ratio. Also, its access time is comparable to that of present day processors and thus it allows them to operate in zero wait state mode. The secondary caches are implemented using very fast SRAMs. The minimum access times of some currently available SRAMs are as low as 2.5 ns [10]. There have been some very exciting developments in the design of DRAMs. DRAMs are coupled with small SRAMs and can be used as cache DRAMs. These cache DRAMs can substitute for secondary caches. They have very low access times (15-20 ns) because of the SRAM at the forefront. The large size of DRAM results in a high hit ratio (around 97%). This module can transfer data at a rate of 1.66 Gbytes/sec, if connected to an on-chip cache via a 64 bit bus [24].

Thus, a two level cache memory hierarchy, local to the CPU, helps substantially in enhancing the performance of present day computer systems by matching the processor speed and improving the overall hit ratio to extremely high levels.

Match the processor and system bus bandwidth

During the 1970's, there was a match between the bandwidth of the fastest microprocessor and that of the bus. Bandwidth is defined as the maximum data transfer rate, expressed as bytes/sec. As times have progressed, performance of microprocessors has improved by leaps and bounds. However, the system buses have not kept pace with the microprocessors. A single CPU board using any of the latest microprocessors, e.g., Intel 80486, Motorola 68040, Motorola 88010, etc. can easily saturate any of the available system buses, e.g., Multibus, VME, ISA, etc.

By *saturating the bus*, we mean that another bus master connected to the same bus would have to wait for a long time to get control of the bus. The system bus has thus become a bottleneck. The examples below help clarify this very important point:

Example 1. If we take an 80386 processor running at 20 Mhz, (cycle time = 50 ns) it takes two cycles in zero wait state mode to transfer 32 bits of data, i.e., 100 ns. This means the processor takes 25 ns to transfer a byte of data, which equates to 40 Mbytes/sec bandwidth. A 32 bit VME implementation with a peak bandwidth of around 40 Mbyte/sec would be just right for this case.

Example 2. A 80486 processor running at 50 Mhz, in a zero wait state, non burst mode would have a bandwidth of 100 Mbytes/sec using the same principle as above. This means for each access to the 32 bit VME bus, the 80486 processor will have to wait for 60 ns. In this case, the 32 VME bus that was enough for a 20 Mhz 80386 would prove insufficient. We need still higher performance buses to match the bandwidth.

In a multiprocessor system implementation, except for the processor currently using the bus, the rest would spend most of their time waiting for their bus arbiter to get control of the bus. We thus have a serious problem to resolve. The following alternatives are available to address the issue of mismatch between processor and bus bandwidth:

New buses with increased bandwidth

If we could keep increasing the bandwidth of the available buses so as to accommodate the ever increasing demands (bandwidth) of the new CPU's and the

peripherals connected to it, we would be fine. We are afraid that this is not going to be the case.

We have come a long way from the days of Multibus [1], which had a bandwidth of 10 Mbytes/sec. The new buses in the market not only have large data buses (32 bits, expandable to 64 bits) as compared to 16 bits for Multibus but also better electrical characteristics. This has resulted in a number of buses with very high bandwidth.

Some of the high performance buses that are used in high performance workstations [22] and have become popular in the recent past are S-bus from Sun Microsystems, Turbochannel from DEC, Micro Channel Architecture bus from IBM and FutureBus+. The maximum bandwidth possible for these are as follows:

TABLE I
BANDWIDTH OF DIFFERENT HIGH PERFORMANCE BUSES

Name	Bandwidth (Mbytes/sec)
S-bus	146 (64 bit version)
Turbo-channel	100 (32 bit version)
MCA	160 (64 bit version)
Futurebus+	100 (32 bit version, can go higher)
VME	80 (64 bit version)

The older 32 bit VME bus from Motorola has been upgraded to 64 bits thereby increasing the bandwidth to 80 Mbytes/sec. Among the newer generation of buses, only S-Bus can currently deliver very high bandwidth; but even it would fail to satisfy the

next generation of processors. The latest microprocessor prototype Alpha from DEC runs at 200 Mhz and thus needs data every $(1/200 \text{ Mhz})$ 5 ns.

As mentioned earlier, the problem only worsens in a multiprocessing environment. It is not possible for the system buses to keep pace with the bandwidth requirements of the next generation processors due to mechanical and electrical reasons. Thus, increasing the bus bandwidth *alone* is not going to solve the problem of finite bus-bandwidth in a multiprocessor system.

Reduce the heavy duty traffic on the system bus

In any computer system, the data movement to/from I/O devices and instruction fetches from the main memory are two main sources of traffic on the system bus. If these fetches can be completed before reaching the system bus and main memory, we would have reduced the processor-memory traffic on the system bus considerably. This would indirectly translate into a substantially higher bus bandwidth.

Example. An 80486 processor is connected to the bus and the bandwidth of both is 100 Mbytes/sec. So long as we have only one processor connected to this bus, no wait states are introduced when the processor accesses the system bus. But, if 10 processors were connected to the same bus, a bus with a bandwidth of 1000 Mbytes/sec is needed to match the bandwidth. This is practically impossible without a great expense.

Instead, if we can reduce the traffic contribution from each processor by 90%, the effective bus bandwidth required for each processor would be 10 Mbytes/sec. The

10 processors would now need an effective bandwidth of only 100 Mbytes/sec which is within the means of the bus.

Local buses and memory (caches) help us accomplish the task of increasing the effective bus bandwidth.

Local bus and memory. The local bus and memory, as the name suggests, is local to the CPU. The local memory bus between the CPU and its memory is extremely fast and can match the CPU's bandwidth requirements. For I/O operations, we have a local I/O expansion bus which helps in keeping I/O transfers local to the I/O master. This also helps in reducing the demands on the system bus and freeing up precious bandwidth.

Primary and Secondary caches. These two caches are the local memory for the processor. The caches, by storing the processors' most recent memory accesses, substantially reduce the processor-memory traffic thereby increasing the effective bus bandwidth, as explained in the example above.

The primary cache, because of its small size, helps in matching the CPU bandwidth, but does not solve the problem of mismatch between the CPU and system bus bandwidth. We need a very high hit ratio cache in between the bus and the processor. This need is fulfilled by the *secondary cache*.

"If one chip uses 50 percent of the bus, a five-chip multiprocessor system should spend most of its time waiting for bus cycles - hardly ideal. It's absolutely essential to use secondary caches with the 486 in multiprocessor configurations, says Gelsinger."

Secondary copy back caches reportedly have scored average hit ratios of 97% for Unix programs and 99% for DOS programs [9]. This has helped manage the demands on bus bandwidth in common bus, shared memory systems.

The two-level private cache hierarchy between the processor and the system bus has thus successfully addressed the two primary performance related concerns of a computer system designer, namely matching the processor speed to that of main memory and also matching the processor and system bus bandwidth.

REDUCE OVERALL SYSTEM COST

The last section covered the details of achieving a high level of performance from the memory system in a computer. The designer has to ensure that this high performance is made available at a reasonable cost. Needless to say, this is not an easy task.

For performance reasons (very low access times), the designer would want the entire memory implemented on the same die as the processor. In this ideal situation, the cycle times are the fastest and also the critical timing paths are very small. However, it would be extremely difficult and expensive to manufacture such a product.

One might want to use only SRAMs instead of DRAMs so as to increase the performance of the memory subsystem. SRAMs are very expensive and the cost of the entire system would thus become prohibitive, if the designer were to use only SRAMs.

Caches once again come to the rescue of the memory system designer. They not only help increase the performance as explained above but also reduce the overall memory system cost by allowing the use of slow speed DRAMs. The two level memory hierarchy, by providing high hit ratios (95-99%), captures most of the CPU's I/O and memory accesses, thereby allowing it to run at *near zero* average wait states. An example should make this point clear.

Example. The two level cache hierarchy provides an overall hit ratio of 98%. A hit in the cache does not result in a wait state for the processor. A cache miss, i.e., the reference has to go to the main memory, incurs 4 wait states. The average number of wait states introduced is

$$(0.98 \times 0) + (0.02 \times 4) = 0.08,$$

which is close to 0.00. An 80386 processor takes two cycles to access a byte in non-pipeline zero wait state mode. It would take $(2 + 0.08)$ 2.08 cycles if it were operating in one wait state mode. The degradation in performance due to a cache-DRAM memory hierarchy is only 4% $((2.08 - 2) / 2)$. The DRAMs introduced a slight deterioration in performance. Thus, a memory system composed of SRAMs alone would have at best provided us with a 4% performance advantage in comparison with a memory system designed using SRAMs & DRAMs.

A typical 66 Mhz Intel-80386 based workstation will have 256-512 Kbyte of cache and 6-8 Mbyte of DRAM. To ensure zero wait state performance, we would need to have 8 Mbyte of SRAMs with a cycle time of around 15 ns. A byte of 15 ns SRAM is much more expensive than a byte of 60 ns DRAM. An SRAM is roughly

an order of magnitude costlier than DRAM according to our preliminary cost comparison (by calling the local distributors). One can easily imagine the substantial cost savings that can be realized by having a SRAM-DRAM memory hierarchy instead of SRAMs.

Importance of simulation in reducing overall system cost

So far, in our discussion about caches, we have given the impression that the designer just needs to connect two levels of caches between the processor and the main memory to solve all the problems associated with processor and memory. Design of a cache sub-system is not as easy as we might have led you to believe. There are a lot of parameters that need to be carefully considered and analyzed with respect to the target computer system and the programs that will be run on it.

The issue here is not just performance but cost/performance ratio. There are several parameters that need to be considered *together* to arrive at the lowest ratio for a given system. We have talked a great deal about cache size. The size is no doubt one of the key parameters but there are others as well, namely placement policy, replacement policy, line size, write strategy, etc. that are part of an effective cache organization.

At times there is a bias towards a certain design parameter, e.g., cache size. Some designers approach this issue with an attitude that the larger the cache size, the higher the performance and thus the better it is, as long as the CPU is running at zero wait states. One is easily tempted to implement the largest zero-wait state cache possible using this approach. Since caches are implemented using extremely fast

SRAMs which are always much more expensive (\$/bit) than DRAMs, implementing the largest cache can prove financially exorbitant.

Also, the largest cache need not necessarily translate into highest *overall* system performance. This is especially true in multiprocessor systems where "A big but dumb cache can perform poorly." [16] It is important that the designer should simulate using all the parameters together and using the software that the target system will eventually execute. This helps in making intelligent decisions. This approach is of immense help in balancing the tricky issue of performance and cost. Let us study a case where simulation proves helpful.

Multi-way vs Direct mapped cache [12]. How does a cache memory designer decide whether to implement a direct mapped cache or an n-way set-associative cache? This is another difficult decision that has to be made and only simulation can provide the answer. There is an inherent bias to design an n-way set associative cache because it offers a higher hit ratio as compared to a direct mapped cache of the same size. But an n-way set associative cache is costlier and more complicated than a direct mapped cache. The reasons are as follows:

n-way ($n > 1$) consumes more memory board space. If $n = 1$, we have the special direct mapped case. For a direct mapped cache, we need only 1 set of cache-tag and cache-data RAM. But, for a 2 way set associative case, 2 sets of cache-tag RAMs and cache-data RAMs are required. Similarly, for a 4 way set-associative cache, 4 sets of cache-tag RAMs are required. Consider, for example, a 4 kbyte main memory, 512 byte cache and line size = 4 bytes.

A direct mapped cache needs 7 address bits ($512/4 = 128$ sets, i.e., 2^7) to index to a block, 2 bits to index a byte within the block and finally 3 bits in the cache tag to identify if the right data address is occupying the block. In a direct mapped case, one set houses only one block. We need a 128x3 (384 bits) cache tag RAM.

A 2-way set associative cache needs 6 address bits ($512/4 = 128$, $128/2 = 64$ sets, i.e., 2^6) to index to a set, 4 bits in each of the *two* cache tag RAMs to identify if the right data address is occupying one of the two possible blocks in the set and finally 2 bits to index a byte within the block. We need two sets of 64x4 (512 bits) cache tag RAM.

Similarly, a 4-way set associative cache needs 5 address bits ($512/4 = 128$, $128/4 = 32$ sets, i.e., 2^5) to index to a set, 5 bits in each of the *four* cache tag RAMs to identify if the right data address is occupying one of the four possible blocks in the set and finally 2 bits to index a byte within the block. We need four sets of 32x5 (640 bits) cache tag RAM.

Thus, a two way cache tag implementation requires two chips (double the memory board space) and a four-way cache tag implementation requires 4 chips (4 times the memory board space) as compared to a direct mapped case. More board space can lead to additional boards and that translates directly to higher costs.

Also, additional boards would lower the performance because of increased delays in the critical system paths. More cache-tag RAMs and cache-data RAMs means more loading on the CPU address pins. A buffer might be needed to prevent the CPU pins from getting loaded, which means that the cache-tag RAM must now be faster by about

7 - 9 ns. The faster the SRAMs, the higher their cost. A direct mapped design implementation does not need additional buffers and the access times of cache tag RAMs are lower than the ones used in the N-way set-associative implementation. This further adds to the cost of an N-way set-associative implementation.

The following passage from [10] should make this point very clear:

"But that doesn't necessarily mean that a 30-ns access time SRAM will work with a 33-Mhz processor," says Sam Orr, SRAM marketing manager for Cypress semiconductor. "The propagation time in the logic to set up and latch the information takes between 7 and 9 ns, so a 25-ns part will just barely squeeze by for a 33-Mhz processor."

A direct mapped cache may be just enough for the performance desired or may be more appealing when compared with the costlier n-way alternative during simulation of the target system. Simulation runs would definitely help in answering questions like: "is the incremental performance using an n-way implementation worth the complexity and additional cost," etc. Only a thorough analysis and simulation can help the designer in striking a good balance between cost and performance and making an intelligent decision.

Example. The caches on Intel 80486 and Motorola 68040 were designed after studying the simulation data that was taken from the customer code. Both the caches are four-way set associative. The end result is very different in that the 80486 processor has 8 kbytes of a unified cache but 68040 has a separate cache for instruction and data.

Caution. While simulating, one must be careful in using traces from an architecture that bears little or no resemblance to the architecture under consideration. Different architectures would probably need two completely distinct cache designs to reach the same goal of lowest cost/performance ratio.

Thus, a multilevel cache hierarchy helps the cache designer strike the right balance between performance and cost. Primary and secondary caches placed between the CPU and system bus increase the throughput and reduce the overall system cost by allowing the use of economical, lower speed DRAMs. A good cache design along with DRAMs would allow the CPU to run at speeds very close to zero-wait state on a sustained basis.

CACHE COHERENCE IN MULTIPROCESSOR SYSTEMS

By providing private (local) memory to each microprocessor we have created a very serious problem. At any given instant of time, multiple processors connected to the same bus would maintain local cached copies of a unique shared memory location. Each processor would then modify its local copy at one time or another. As a result, an inconsistent view of this particular shared memory location is projected across the system. This is commonly referred to as the cache coherence problem.

Cache coherence schemes (protocols) ensure that each request from a processor gets the most up-to-date copy of the block. There are a lot of different ways one can achieve this end result. Each strategy has distinct advantages and limitations. Some

of these cache coherence enforcement schemes are ideally suited for a particular architecture while others may not work at all.

The section below discusses the most important issues that a designer needs to consider before choosing a protocol.

Important issues concerning coherence protocols

The issues listed here are the fundamental ones involved in the design and selection of a cache coherence protocol [19]:

Correctness of the protocol. This is the most important issue concerning a coherence protocol. How does one ensure that a read request by one processor returns the most up-to-date copy of the data in the system.

This is not an issue in the case of a single bus based multi-processor system, because the bus allows only one processor to access the data, in other words, the bus is the serializing point. However, in the case of systems designed around inter-connection networks, multiple processors are simultaneously reading or writing into the same block, making it very difficult to ensure correctness.

Protocol complexity. The protocol should not be very complex in its implementation. The complexity, performance and correctness issues are very closely related. The protocol can become increasingly complex to ensure correctness, as in the case of systems based on inter-connection networks.

If the protocol is too complex, it means that it is also very difficult to implement. A complex implementation results in poor performance because the latency of memory requests would increase to ensure correctness.

A protocol should be simple, which translates to ease of implementation and higher performance. This finally translates into lower cost of implementation. Snoopy cache protocols on bus-based systems are relatively simple, easy to implement and hence the most popular on commercial machines.

Overall system performance. The memory latency or miss penalty and bandwidth are the big factors in controlling overall system performance.

The protocol should be scalable when more processor-memory pairs are added to the system. The bandwidth should not become the limiting factor when the size of the system becomes large, as in the case of a single bus based system.

Whenever the processor reference encounters a miss in its cache, the protocol should ensure that the time required to service the miss is as small as possible. Large multilevel caches and hierarchical cache hierarchies play a big role in reducing the miss penalty.

CACHE COHERENCE SOLUTIONS IN MULTIPROCESSOR SYSTEMS

The solutions can be categorized as hardware or software-based. The software-based solutions (protocols) do need some hardware support to maintain consistency. Hardware-based protocols ensure that the software always sees a coherent view of the data block across the system.

The hardware and software-based protocols differ primarily in how they determine whether the block is shared, how they find out where the blocks reside, and how they invalidate or update copies in the caches and main memory.

Hardware-based protocols

Snoopy cache protocol (SCP). In a snoopy cache protocol, each cache controller monitors the transactions of all other caches on the system bus. In other words, each cache controller snoops the bus to detect any coherence related activity. If a bus transaction threatens the consistency of a locally cached data block, the controller can take appropriate action, e.g., invalidate its copy of the data.

This protocol generates a lot of traffic on the network. This brings to life the issue of *bus bandwidth*. As the number of processors connected to the bus increase, so does the traffic on the network. The bus should be able to handle the increased demand on bandwidth as more processors are added. Since the buses have a fixed bandwidth, this does limit the *scalability* of the design [4].

SCP is better suited for a single bus based system (with processors and memory sharing the same bus). It is easy to broadcast the message on a common shared bus and also easier to monitor the bus activity.

A good example of a commercial system using this approach is the Sequent Computer Systems' Symmetry. The Symmetry system allows up to 30 processors, each with 64 Kbytes of 2-way set associative, write-back cache connected over the shared system bus. The cache controllers snoop the bus to maintain cache coherence [2].

Snoopy protocols' ease of implementation has made them one of the most popular in the industry and this can be seen from the number of commercial implementations adopting this strategy.

Directory based schemes (DBS). Snoopy protocols have two serious limitations, i.e., limited bandwidth and scalability. These limitations must be overcome if one has to design larger systems delivering very high performance.

Instead of using a common bus as an interconnection network, one can use other forms of inter-connection networks whose bandwidth is scalable, e.g., multi-stage networks. The network traffic generated by the snoopy protocol due to broadcasts made at times of invalidation or updates needs to be eliminated or at least minimized. In the case of directory based schemes [2], an attempt is made to eliminate the broadcasts. The need for broadcasts in the case of SCP arose because the initiating cache did not have any information about the location of cached data blocks that had to be updated or invalidated. To eliminate broadcasts, the cache must know the precise location of the cached data blocks in the system. Then the communication can be limited to the caches that have a copy of the block.

The location information, along with the state of the cached data blocks, is stored in an entity known as the directory. Directory based schemes keep a *separate directory* associated with main memory that stores the state and location of each cached data block in the system. This directory may be kept in a centralized location or distributed along with the different memory modules in the system.

The location information points to the caches that contain the data. For one of the implementations, known as the *full map directory scheme* the presence information is typically a bit map, where each bit corresponds to a processor in the system. If the

bit is set, the cache associated with this processor has a copy of the data block and vice-versa.

In the full map scheme, any cache can store any copy of the block. This is because of the one bit per processor entry in the directory. The cost one pays for this is that the size of the directory grows as the number of processors increase.

The rate of growth of the directory can be checked by limiting the number of cached copies that can reside in the system. This is called the *limited directory* scheme. If a system has 'n' processors (n is a power of 2), each pointer in the directory would need $\log_2(n)$ bits. If we have a 16 processor system, each pointer would need four bits, and say we can have 3 copies in the system. If a fourth processor wants to cache the block, the main memory invalidates the contents of one of the 3 caches and then the block is loaded. The directory based approach is significantly different from the snoopy cache protocol, in that the location of the caches that have a copy of the shared data is known. Instead of broadcasting messages, directed messages are sent to only the caches that have the particular data block in them.

The advantages of this design are as follows:

- a. one has the flexibility of choosing an inter-connection network as compared to a Snoopy protocol which forces one to use a common bus.
- b. it is possible to *scale* cache coherent multi-processors to a large number of processors-memory pairs, e.g., the Stanford Dash multiprocessor is a scalable architecture and employs a distributed directory scheme. [19]

c. network traffic is significantly reduced because of the directed messages to only the caches that have a copy of the block in question.

The disadvantages of using this approach are as follows:

a. the size of the directory memory reaches unmanageable proportions in the case of a full map scheme if the number of processors increase beyond a certain point.

b. the design is scalable only to a point. Once the design is completed, it is not possible to add more processor memory pairs.

For example, if the system is designed using the full map scheme for 48 processors, then there are 48 presence bits associated with each block in the main memory. Now, if we had to upgrade the system to a 60 processor machine, the memory cards will have to be changed to reflect 60 bits in the directory memory. Additional wiring will be needed, too.

The same is true for the limited directory scheme, the pointer size changes when the number of processors increase, i.e., for 9-16 processors, we need four bits in the pointer but for 17-32 bits, we need 5 bits.

Cache coherent network architectures (CCNA). Cache coherent network architectures hold the most promise for very large scale, shared memory, multi-processor based systems. The reason is they address the weaknesses of both the single bus based and directory based designs. The directory based systems, even though scalable, have problems in implementing the design when the number of processors is too large. The hardware becomes too complex to design and implement on that large a scale.

Cache coherent network architectures [2] employ a hierarchical bus structure. Snoopy coherence protocols are employed because of their ease of implementation. The hierarchical cache/bus architecture reduces the network traffic due to the protocol but at the same time is easy to implement as compared to DBS. The architecture is also scalable to a very large number of processor-memory pairs, but to a finite limit.

The scalability of the CCNA architecture is limited by the system bus bandwidth and the electrical loading characteristics. In the case of a full map directory scheme, the scalability is limited by the hardware design; the number of bits in the presence vector at the time of design decides the number of processor memory pairs we can have in the system.

The hierarchical architectures use Multi-level caches which also helps in reducing the network traffic. Later in the chapter, we discuss more about the impact of multilevel caches on coherence. Multilevel caches should prove a big help for the cache coherent architectures. These architectures are still in the research and development phase and it remains to be seen whether they would provide a meaningful improvement over the existing designs in a real life environment.

Software-based protocols

Software-based protocols [2] attempt to reduce the network traffic and also provide an economical solution to the cache-coherence problem. These protocols do need limited hardware support, but the hardware required is much simpler as compared to hardware-based protocols.

In the section on hardware-based protocols, we discussed the need of a coherence protocol to take care of inconsistent copies of data blocks in the caches. Inconsistent copies arise because these data blocks are shared read/write in nature and all processors are allowed to update them.

However, if the blocks in the caches are never inconsistent, we would never need any of the hardware-based coherence protocols discussed earlier. This is the principle behind the software-based coherence schemes. The compiler ensures that an inconsistent copy of a block would never reside in the memory system.

The compiler needs hardware support to ensure that the caches never have an inconsistent data block. It decides which cached data blocks need to be invalidated or declared uncacheable to maintain coherence across the system. This decision is made prior to run time, i.e., during compilation. This is the main difference as compared to hardware-based protocols.

Limit caching of shared read/write data blocks [2]. The compiler analyzes the program and marks the data as cacheable or non-cacheable. During safe times, all the processors are only going to read the cached read/write data block or it is going to be updated by only one processor. Under these circumstances, it is safe to declare this block as cacheable.

An example of a safe time would be the execution interval of a critical section; during this period only one processor can update a shared read/write data block, making it safe to be cached. After the execution is over, other processors might want to write

to this block, hence the main memory is updated (can use write-through) and the block is invalidated from the cache to ensure that main memory is consistent.

Another method to determine Cacheability [2]. The compiler determines the cacheability of blocks by statically partitioning the data structure into different computational units. The reference marking is based on the partitioning process. Access to a shared variable is determined by the computational unit to which it belongs.

For example, if the data block belongs to a computational unit described as *read only* by an arbitrary number of processors, it can be safely cached. On the other hand, if it belongs to a computational unit that is *read-write* for an arbitrary number of processors, then the block should not be marked cacheable.

The compiler's task is to analyze the data dependencies and generate appropriate cache instructions to control the cacheability and invalidation of shared data.

The performance of the system depends on the performance of the compiler. This implementation needs very simple hardware support. It does not generate heavy network traffic. However, there is no commercial implementation to date that employs this principle.

MULTILEVEL CACHES AND THEIR IMPACT ON CACHE COHERENCE

A multilevel cache hierarchy helps in improving the overall hit ratio of the system and also the average access time. But it complicates the issue of cache coherence. With multi level caches, we need to maintain a coherent copy in all the levels of the memory hierarchy.

For instance, when there is a bus-write cycle on the bus in progress, which means a processor (the current bus master) is updating its copy of a shared read/write block, all caches except the one that belongs to this bus master need to invalidate their copy, if present. Since there is no way of knowing whether a cache at a certain level has the copy or not, the simplest way would be to send invalidation request all the way down to the lowest level in the hierarchy, i.e., to the cache connected to the processor.

The primary cache would now be exposed to all invalidation cycles appearing on the system bus. This means that whenever there is an invalidation cycle in progress, the processor will have to be stopped until the invalidation is over. Worse still, the processor may have to stop even though its cache does not have a copy of the particular data block.

This approach is understandably very inefficient and needs to be modified. The primary caches' responsibility is to keep the processor fed with data and stop it only when the cache indeed has a copy of the block that needs to be invalidated. Also, the secondary cache stays relatively idle (it is used for 4 - 8% of the references that suffer a miss in the primary cache). If the secondary cache can somehow filter the incoming invalidation signals and send only the genuine ones to the primary cache below, it would lead to a big improvement in performance.

Multilevel inclusion principle (MLI)

"The principle of inclusion is a method by which a secondary cache can be used to screen invalidation cycles, thereby limiting the number of bus invalidations which are passed through to the primary cache." [12]

The basic operation can be described as follows: A reference made by the processor is satisfied by the cache closest to the processor that had a hit. When this cache is supplying data to the processor, it also supplies a copy of the data to all the caches that lie in between itself and the processor.

"On the basis of this model, we say that a multilevel cache hierarchy has the inclusion property if the contents of a cache at level $i+1$, L^{i+1} , is a super set of the contents of all its children caches, L^i , at level i ." [6]

Impact of MLI on Cache coherence protocols. If MLI is implemented in a multilevel cache hierarchy, the cache coherence protocol will become easier to implement. This is because each of the local secondary level caches will have a copy of the data present in the primary cache. Thus only the secondary caches need to be snooping.

This helps in two ways: [25]

a. The processor-onchip cache pair is shielded from the invalidation traffic except for the genuine case, where the onchip cache does contain a shared block that needs to be invalidated.

b. Since the intelligence for implementing the coherence protocol now lies at the secondary cache, the design of the onchip cache becomes simple resulting in faster access time.

The disadvantages of enforcing MLI in the system are:

a. the size of higher level caches grows very large.

b. as the size of the caches increase, so does their access times, making them slow.

c. the hit ratio at the higher levels goes down, because of the restrictions placed due to MLI.

CHARACTERISTICS AND LIMITATIONS OF SHARED BUS SYSTEMS

The shared bus is a very attractive type of inter-connection network because of its simplicity. It is also easy to implement cache coherence protocols on a shared bus system. All the cache controllers can monitor the activities of each other and take action in response to a bus transaction, if warranted.

However, this implementation strategy has the drawback of *limited bandwidth and scalability*. This means we can only connect a certain number of processors. The electrical characteristics, e.g., bus loading due to multi dropping are directly responsible for the limited scalability of this approach. We need to consider the alternatives suggested earlier in the section on cache coherent network architectures, if we have to move towards larger and higher performance systems. These are covered in the section below.

Evolution towards multi-level caches and multiple bus based microprocessor systems

One solution to overcome the limitations mentioned above can be to incorporate several parallel, independent buses in the system. This implementation will be capable of handling several memory requests concurrently. The independent buses coupled with multi-level caches would help us match the processor speed to the main memory speed,

match the processor bandwidth to system bus bandwidth, reduce the overall system cost while providing high performance.

Also, there does not seem to be much more room for *architectural improvement* in the design of a single shared bus based multi-processor system. What we mean is the system architect does not have the luxury of making significant changes to the basic architecture itself. The improvements in performance can come only by upgrading the system with faster processors, memory chips and system buses.

We have chosen a multi-level, multiple bus based, hierarchical shared memory system as the architecture of choice for my thesis. The architecture is defined and explained in great detail in the next chapter. The reasons for choosing this type of an architecture are:

- a. intuitive appeal and simplicity of this architecture [5]
- b. extension of shared common bus architecture
- c. snoopy cache coherence protocols are simple and efficient in their implementation [2]
- d. has more room for modularity and scalability as compared to a single bus-based design
- e. provides an elegant programming model [19]
- f. capable of delivering very high performance [5]

CHAPTER III

TREEBUS ARCHITECTURE AND CACHE COHERENCE PROTOCOL

The last section in Chapter 2 briefly outlined the reasons for selecting a hierarchical, shared memory multiprocessor system as the architecture of choice for this thesis. The main reasons are that this is a scalable architecture, capable of delivering very high performance and is also easy to program (due to the shared main memory). "The shared-memory paradigm has the advantage that the programmer is not burdened with the issues of data partitioning, and accessibility of data from all processors simplifies the task of dynamic load distribution." [19]

Several shared memory based multiprocessor systems have been discussed in current literature [5,6,13] but we will focus on the architecture first proposed by Wilson [13] and later analyzed by Jog [5]. This architecture is called TREEBUS [5,15]. The main memory is at the root of the tree, the branches are the buses, and the processor-cache pairs form the leaves.

Another salient feature of this architecture is that it uses an efficient snoopy cache protocol to maintain coherence across the caches in the system. The buses in the system enable us to use this easy to implement cache coherence scheme.

This chapter consists of three major parts, the first section explains the TREEBUS architecture in detail, the second section covers the implementation of the cache directory and the last section covers the coherence

protocol in detail.

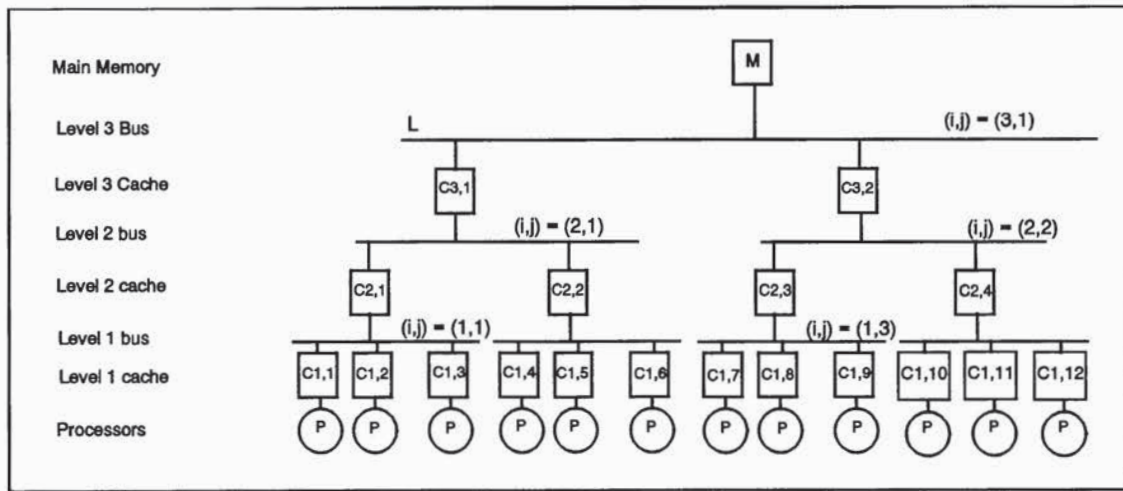


Figure 1. The TREEBUS architecture.

TREEBUS ARCHITECTURE

The TREEBUS architecture is shown in Figure 1. There are multiple buses arranged in a hierarchical fashion. All the buses operate independently of each other. Buses at two consecutive levels, say level one and two or level two and three, are connected to each other using caches. In other words, we also have a hierarchical cache memory organization.

The architecture can be defined using the following variables:

1. L , number of levels in the hierarchy. Level refers to the depth of the tree structure.
2. i , a level in the hierarchy, under consideration at a particular instant
3. N_i , the number of buses at level i

4. (i,j) , bus j at level i in the hierarchy,
5. n_{ij} , the number of caches connected to bus j at level i
6. $C_{i,q}$, a cache connected to the (i,j) bus, where i, j , and q are defined in equation 1.

$$\begin{aligned}
 1 &\leq i \leq L \\
 1 &\leq j \leq N_i \\
 1 &\leq q \leq (n_{ij} \times N_i)
 \end{aligned}
 \tag{1}$$

The processors P are connected to their own private caches which are at level one. The boxes with $C_{i,q}$ represent the caches in the system. Each bus (i,j) is connected to caches at two different levels, i.e., i and $i+1$. A level 1 bus is connected to a group of $C_{1,q}$ caches and a level 2 cache, $C_{2,q}$. The main memory M is connected to the bus at level L and can also be thought of as a level $L+1$ cache. Each group of processor-memory pairs connected to the level 1 bus is referred to as a cluster [13] or a super-processor [5].

These clusters form the building blocks of the TREEBUS architecture. The cluster architecture is the same as a standard shared memory, common bus, multiprocessor system, e.g., Sequent Balance or Symmetry [2]. The difference here is that each cluster is connected to other clusters in the system via a cache memory at the next higher level in the hierarchy instead of being connected directly to the main memory as in the case of a Sequent system.

A level 1 cluster connected to the level 1 bus is connected to a level 2 cache. A number of these clusters are connected to the level 2 bus via their respective level 2 caches. This bigger block can be called a super cluster or a level 2 cluster. In our example, we have 4 super clusters at level 2 and 2 super clusters at level 3. The process of making bigger clusters is recursive and we can continue until we reach the highest level in the hierarchy.

Note: Throughout this chapter, all examples on TREEBUS architecture will refer to Figure 1 unless otherwise stated.

Some definitions as applied to the TREEBUS architecture

Tree, root and leaves: A treebus system is a tree with main memory at its root. The entire system, complete with processors, caches, buses and main memory forms the tree. The processors are the leaves.

Sub-tree: A sub tree is a part of the tree with a cache as the root instead of main memory, e.g., a cluster with a level 2 cache, $C_{2,1}$ as the root is a sub-tree. We have 2 sub-trees at level 3 and 4 at level 2 in the system.

Branches: The buses at different levels are the branches. The maximum number of branches at level i is N_i .

Branching factor: The number of caches connected to the bus j , at level i , is called the branching factor $n_{i,j}$, e.g., the branching factor for each of the buses at level 3 is 2.

Symmetric system: The number of caches connected to all the buses at a particular level i is the same in a symmetric system, i.e., $n_{i,j}$ for all j is a constant. The

architecture in Figure 1 represents a symmetric system. There are two buses at level 2 and they both have a branching factor of 2. All the level one buses have 3 caches connected to them.

Peer cache: For any cache $C_{i,q}$, connected to bus j at level i , all the remaining $n_{i,j-1}$ caches are referred to as peer caches, e.g., a cache connected to the level 3 bus has 1 peer cache.

Descendant caches: For a sub-tree at level i with cache $C_{i,q}$ as the root, all the caches at levels lower than i that consider this cache as the root are its descendants, e.g., for cache $C_{2,1}$ which is the root of a sub-tree at level 2, all the caches at level 1 that consider $C_{2,1}$ as their root (3 in all), are descendants of $C_{2,1}$.

Parent cache: In the explanation for descendant caches above, the cache that is at the root of the sub-tree is also referred to as the parent cache. $C_{2,1}$ is the parent cache for the sub tree consisting of $C_{1,1}$, $C_{1,2}$ and $C_{1,3}$ caches. $C_{2,1}$ is at the root of this sub-tree. Similarly, main memory M is the parent of all caches in the system.

Block size: The size of the block at level i , b_i , over which coherency is maintained is the block size. The block size is kept the same as the transfer size. This simplifies the management of the coherence protocol and also the data transfer process.

The architectural details for the symmetric TREEBUS architecture shown in Figure 1 are summarized in the table below:

TABLE II
CHARACTERISTICS OF TREEBUS ARCHITECTURE IN FIGURE 1

Number of buses at level 1, N_1	4
Number of buses at level 2, N_2	2
Number of buses at level 3, N_3	1
Branching factor at level 1, $n_{1,j}$	3
Branching factor at level 2, $n_{2,j}$	2
Branching factor at level 3, $n_{3,j}$	2
Number of levels, L	3
Number of clusters at level 2	4
Number of clusters at level 3	2

Cache coherence

The protocol that maintains coherence across all levels of the hierarchy is based on the Snoopy-cache principle. The TREEBUS architecture employs the Multi Level Inclusion principle to simplify the implementation of the coherence protocol and increase the efficiency of the entire system. Without MLI, the implementation of this architecture would be highly inefficient.

A large level $i+1$ cache would be able to satisfy most of the memory requests (high hit ratio) from levels below. A very high hit rate at level $i+1$ means lower network traffic on the $(i+1, j)$ bus which leads to higher effective bus bandwidth. This allows more processors to be connected to the bus and finally all these factors contribute towards still higher performance.

The minimum size of the cache at any level can be calculated by continuously applying equation 2. The size of cache at level 1, $C_{Size,1}$ must be known before using the equation.

$$C_{Size,i+1} = (n_{i,1} \times C_{Size,i}) \quad (2)$$

The size of caches at each level in the hierarchy should follow a rule of thumb as suggested by Wilson [13], i.e., a cache at level $i+1$ should roughly be an order of magnitude larger than the sum of the size of all descendant caches at level i .

The expression in equation 2 on the right hand side can be multiplied by a factor α to incorporate this suggestion. Please refer equation 3 below. If α is 1, then we have equation 2, which says that the size of a cache at level $i+1$ is the sum of all its descendant caches at level below. Increasing the value of α leads to a higher hit ratio which leads to higher bandwidth and hence higher performance.

$$C_{Size,i+1} = (n_{i,1} \times C_{Size,i}) \times \alpha \quad (3)$$

Applying the rule of thumb leads to very large sizes for the higher level caches and the main memory in the system. Assuming $C_{Size,i}$ at each level i is same, $C_{Size,1}$ is 256 Kbytes, and $\alpha = 10$, the size of memories at levels greater than 1 can be calculated

using equation 2. $C_{Size,4}$ refers to the size of main memory which is connected to the level 3 bus.

The size of an individual cache at various levels in the system are as follows:

TABLE III
SIZE OF INDIVIDUAL CACHES AT EACH LEVEL

Level i	$C_{Size,i}$ (bytes)
1	256 Kbytes
2	7.5 Mbytes
3	150 Mbytes
4	3 Gigabytes (main memory)

The size of all the caches at level 2 is 7.5 Mbytes each and at level 3 is 150 Mbytes each. The total memory required for the sample symmetric system is given by equation 3:

$$Totalsize = \left(\sum_{i=1}^{i=L} (N_i \times n_{i,1}) \times C_{size,i} \right) + C_{Size,L+1} \quad (4)$$

The product $(N_i \times n_{i,1})$ gives the total number of caches at a particular level i. The total memory needed for the system is 3333 Mbytes or 3.333 Gigabytes. As can be seen from equation 3 and the table above, the memory size at higher levels and the total memory size for the system grows exponentially very rapidly.

However, present day memories have become more and more dense, e.g., CYM1841, a high-density 8M-bit SRAM module with an access time of 20 ns (organized as 256K words by 32 bits) is now available from Cypress Semiconductor [14].

DRAMs are also available with package densities of 16Mbit x 1 and access times around 50 ns, with 64 Mbit x 1 DRAMs to follow suit in the near future. Hitachi showed a prototype of a 64M-bit x 1 DRAM in mid 1990 and plans to start volume production sometime in 1995 [21]. Hence, designing a memory system for this architecture should be feasible.

For example, in Intel's Paragon_{TM} XP/S system [20], each node (processor) can have 16-128 Mbytes of main memory, 2-128 Mbytes of cache and there can be as many as 1000 nodes which translates into 128 Gbytes of main memory. This is made possible by the high density, high speed memory modules currently available in the market [20].

However, in case the designer faces problems due to the large size of these memories, the rule of thumb can be relaxed which should restrict the size explosion of the higher level caches. This might lead to more invalidations percolating to the level 1 cache due to replacement of blocks at higher levels. We are not going to explore the optimum size of the cache that leads to the highest performance. We would however like to determine the cost of the memory to achieve a particular level of performance.

Data movement in the hierarchy -- an overview

The data from the main (shared) memory reaches different caches in the system in response to a processor's read or write request, called a P_read or P_write request.

P_read request. In response to a processor read request, the level 1 cache tries to supply the data. If it does not have a copy (miss), the request appears on the level 1 bus as B_read. An unsuccessful P_read request to level 1 cache has now been transformed into a B_read request on the level 1 bus. Caches that might respond are the peer caches of the cache that initiated the B_read request.

Caches at level 1 try to satisfy the request, but if they do not have a copy, then the request moves up to the parent cache of the level 1 cache. The unsuccessful B_read request on the level 1 bus is transformed to a P_read request for the level 2 cache. This cache either supplies the data or sends the request to the level 2 bus as B_read, in case of a miss.

Recall/Write back. If a peer cache's copy is not an up-to-date one, it is updated by recalling the up-to-date copy from one of its descendants. Once the peer cache has updated its copy, it can go ahead and supply a copy to the requesting cache. The requesting cache, if full, would need to create room for the incoming block by replacing an already existing block. If this block was modified in the cache, it needs to be written back into the next higher level cache. This will ensure consistency of data across the system.

If none of the caches in the system have a copy of the block, the request finally reaches the main memory. Main memory supplies the data to the level 3 cache. The

level 3 parent cache supplies the data to its descendant at level 2 which in turn supplies it to its descendant at level 1. The data finally reaches the level 1 cache which supplies it to the processor.

Thus, a P_read request to the level 1 cache can manifest itself as multiple P_reads and B_writes. A P_read request to a level i cache, $i > 1$ means that none of the descendant caches at levels one through $i-1$ have a copy of the data. A B-read transaction on the level (i,j) bus is caused by a read miss in one of the level i caches and all its descendants. An unsuccessful P_read at level i gets transformed into a B_read at level i and if this B_read suffers a miss, it is transformed into a P_read request for the level $i+1$ cache.

If there are multiple copies of a block at different levels in the system, several actions must occur before a P-write request can update the block in the cache. Consider a P_write to cache C1,3. We need to ensure that only the parents of C1,3 have a copy, i.e., all the peer caches and descendants of C1,3 and its parent caches need to invalidate their copies. This is to ensure that no two processors update the same block at the same time. This action ensures coherency of data across the system.

A P_write request to the level 1 cache gets transformed into a B_write request on the level 1 bus and a P_write request to the level 2 cache, parent to the level 1 cache. In response to a B_write, all the peer caches on the bus with a copy invalidate it.

The P_write request at level i propagates as P_write to level $i+1$ until it reaches the top of the hierarchy or it is certain that there is only one copy in each of the levels

above i . A Bus write on a level (i,j) bus, where $i > 1$, also gets transformed into B_write requests on the levels below in the hierarchy to invalidate all the copies in the descendants of the peer caches that responded to B_write.

Finally, we are left with only one copy of the data at each level and this copy resides in the parent caches of C1,3. Thus a P_write request to a level 1 cache gets transformed into multiple P_writes and B_writes. P_writes propagate up the hierarchy only but B_writes propagate both up and down in the hierarchy.

CACHE DIRECTORY ORGANIZATION

States of Cached blocks

In a typical uniprocessor, multilevel cache environment, the states associated with a cached block are clean, dirty and valid. The TREEBUS is not only a multilevel but a multiprocessor system as well. Some new scenarios that must be handled are as follows:

i. A cache places a request for a block on the level i bus (B_read). If more than one peer cache has a copy of this block, which one should respond to the bus request?

If one and only one of these caches is made responsible for supplying the data, it would fix the problem. This is the principle behind the concept of *block ownership*. If multiple caches connected to a level (i,j) bus have the same copy, only one of the caches is marked as the owner. This cache is now responsible for supplying a copy of the block when there is a bus read request for it.

ii. We fixed the ownership problem, but what if the cache that responds to the request does not have the most up-to-date copy of the block?

This raises the concept of *consistency*. By consistent we mean that the cache has the most up-to-date copy of the block. In this case, the owner cache has to get the most up-to-date copy of the block and then supply it to the requesting cache. However, if the owner cache's copy had been up-to-date, it would have supplied the block right away.

iii. Does the meaning of clean and dirty state stay the same as in the uniprocessor case?

In the uniprocessor case, if a block is clean, it can be replaced without performing a write-back. A dirty block, on the other hand, must be written back before being purged. A write to a block in state clean changes its state to dirty. A write to a dirty block does not cause any change in state. In the multiprocessor case, the same meaning as above holds but some more information is added. A block in state clean also means that there can be multiple copies of this block in the system. Hence, a P_write to a clean block gets transformed into multiple P_writes and B_writes, as explained earlier in the sub-section Data movement in the hierarchy-an overview.

If the P_read request suffers a read miss, the request is sent on the bus to see if one of the peer caches can supply a copy instead of directly sending the request to its parent cache as in a uniprocessor case. This is done because one of the peer caches can have a copy of the data.

For the architecture under study, we need to know the ownership and consistency information to take care of bus requests and clean/dirty information for handling processor's requests.

The following 3 attributes are critical, namely owner/non-owner, consistent/inconsistent and clean/dirty. These three attributes taken together define the state of a block completely. Let us explain these attributes in more detail:

Clean. A clean block at level i in the memory hierarchy is up-to-date with respect to the block at level $i+1$. A P_read request for a block absent in the system brings it in state clean. Subsequent read requests to this block do not alter its state and the requests are satisfied by the cache without informing other caches.

There can be multiple copies of a clean block at level i . If there exists a clean block at level i in the system, then there can only be clean blocks at levels lower than ' i ' in the hierarchy. Hence, if a processor writes into a clean block, other caches in the system have to be told to invalidate their copies.

Dirty. If the copy of a data block at level $i+1$ in the memory hierarchy is stale with respect to the copy at level i , then the block at level i is in state dirty. There can be only one copy of a dirty block at each level in the hierarchy because of consistency reasons. This ensures that at any given instant of time, no two processors in the system can update the same copy. If a dirty block has to be purged, it must be written back.

Consistent. A consistent block is the most up-to-date copy in the system. The cache with a copy that is closest to the processor must have the block in state consistent. The opposite of consistent is *inconsistent*.

There can be more than one copy of a consistent block among caches connected to a bus(i,j). However, if a block is in state inconsistent, only one of the caches connected to this bus can have the data block. A block marked inconsistent has to be dirty as well.

A clean block is always consistent, but the reverse need not be true. For example, a P_write request to the cache at level 1 marks the block as dirty and consistent. The block is dirty because after the P_write, the copy at level 2 will be out of date with respect to the copy at level 1. Since the level 1 copy is the most current copy, it is termed as consistent.

Owner. As explained earlier, at most one cache can be the owner of a block for each bus. For example, if C1,1 and C1,3 share a block, either C1,1 or C1,3 can be the owner, but not both. However, there can be multiple owners at the same level, if the caches connected to the buses at the same level have a copy of the block.

The Owner cache is always responsible for supplying an up-to-date copy of the block. If an owner cache has a stale copy, it first updates its copy and then supplies the requesting cache.

Since a dirty block is the only copy at a level, say i, the cache containing it has to be the Owner of the block to respond to future bus requests for the block. Similarly, a block in state inconsistent is also the only copy at level i and has to be the owner as well. This will ensure that the cache containing this block can update its copy from one of its descendants in response to a future bus request for the block and then supply a copy to the requesting cache.

Non-owner. Since the cache is not the owner of the block, it does not respond to a bus read transaction even if it has a copy of the block. In response to a bus-write transaction, the cache invalidates its copy and ensures that all copies of the block in its descendants are also invalidated.

We thus have three attributes that transform into eight possible combinations. Out of the possible eight, only four are permissible. The table below shows all the combinations possible and the ones that are permitted to occur.

TABLE IV
ATTRIBUTES ASSOCIATED WITH EACH STATE

No.	Owner/Non-owner	Consistent/ Inconsistent	Clean/ Dirty	Accept/ reject
0	Owner	consistent	clean	accept
1	Owner	consistent	dirty	accept
2	Owner	inconsistent	clean	reject
3	Owner	inconsistent	dirty	accept
4	Non-owner	consistent	clean	accept
5	Non-owner	consistent	dirty	reject
6	Non-owner	inconsistent	clean	reject
7	Non-owner	inconsistent	dirty	reject

Invalid. We have not talked about this state so far. All the four acceptable states above must have the data in state *valid*, i.e., the data is usable. If a block is in state *invalid*, it is the same as being absent from that location. A processor request to an

invalid block generates a miss. Also, the cache with an invalid copy does not respond to any bus transaction pertaining to the block.

Thus, the following five states should completely specify any cached block in the system:

1. Owner, consistent, clean
2. Owner, consistent, dirty
3. Owner, inconsistent, dirty
4. Non-owner, consistent, clean
5. Invalid

Explanation for unacceptable states. States 2 and 6 in the table above imply that the block is inconsistent and clean. By definition, a clean block is always the most up-to-date and an inconsistent block is just the opposite, hence this state is not permitted.

States 5 and 7 are unacceptable because of *ownership* reasons. The block is dirty in state 5 and inconsistent in state 7. An inconsistent block must also be dirty. By definition, a dirty or an inconsistent block is the only copy at that level. If there had been multiple copies, then one of the other caches connected to the same bus would have taken care of future bus requests. But since this is the only copy, the cache containing this block must take responsibility for supplying a copy in response to read requests from the peer caches, i.e., the block must be in state owner.

In other words, an inconsistent or a dirty block must be the owner and a non-owner block can only be in state clean and consistent as in state 4.

Implementation of Cache directory

The purpose of the cache directory in a level i cache is to respond to processor requests from level $i-1$ and bus requests on the level i bus. It uses the state information associated with the block to decide the right course of action. Needless to say, the state information resides in the cache directory.

For a processor request, the type of action taken by the cache directory depends on whether the block is in state invalid or clean/dirty. The response to a bus request is based on the ownership, consistency and validity information. The cache directory serves two distinct functions, i.e., it responds to processor and bus requests. We can implement two different directories to take care of these distinct functions. The two directories can be known as 'Processor directory' and 'Bus directory.'

The separate directories are also important from a performance point of view. When the bus directory is responding to a bus transaction, the processor directory need not wait and can respond to the requests made by the processors from the levels below. A single directory would adversely affect the performance of the system. In case of only one directory, the directory can easily become a bottleneck.

However, if the state of a block has to be modified in response to either a processor or bus request, both of the directory entries are updated in a single atomic operation.

Processor directory. The processor directory at level i , as the name suggests, takes care of the processor requests from the level i cluster. In response to a processor request, it determines whether there is a valid copy in the cache, i.e., if the request is

a hit or miss. If the request is a hit, the block is supplied, otherwise, the request is sent to the level (i,j) bus. The states implemented in the processor directory are:

1. clean
2. dirty
3. invalid.

Bus directory. The bus directory on the other hand, helps the cache to take care of the following:

1. invalidate its own copy and the copies in its descendant caches in response to a bus write transaction.
2. supply a copy of the block in response to a bus read transaction, provided it is the owner of the block.

There are four states associated with a cached block in the bus-side directory. They are as follows:

1. Owner, Consistent: The corresponding state in the processor directory can be clean or dirty.
2. Owner, Inconsistent: When a processor writes to a clean block, the copy in level 1 cache is in state Owner & consistent, but the copy in the parent caches is marked as Owner & inconsistent. This is because the higher level copies of the same block are out-of-date with respect to the copy at level 1.
3. Non-owner, consistent: The block is consistent with other copies at level i. The corresponding state in the processor directory is clean.

4. Invalid: The cache with an invalid copy ignores the bus transaction because it need not do anything with respect to the block.

CACHE COHERENCE PROTOCOL (IN DETAIL)

State transitions

Figure 2 shows how the five types of transactions affect the states of cached blocks in the processor and bus side directory. They are processor read/write requests, bus read/write requests and write-back/recall requests.

This state diagram refers to the state of the block in a cache connected to the level (i,j) bus. The diagram in Figure 3 is a special case of the one in Figure 2 and refers to transitions for a cached block connected to the level $(1,j)$ bus.

A processor request to a level i cache means that the request is coming from the caches at level $(i-1)$ whose parent is this cache at level i . A request on bus j at level i is initiated by one of the caches connected to the level (i,j) bus.

Let us now study the effect of these transactions on each of the five cached block states in detail.

Owner, consistent, Clean. A processor read request to a block that is absent in the system causes a read miss. The P_read request traverses all the way to the main memory. The block is supplied by main memory to the level 3 cache, which passes it on to the level 2 cache, which in turn passes it to the level 1 cache. The level 1 cache finally supplies the block to the processor. The block now exists in all the

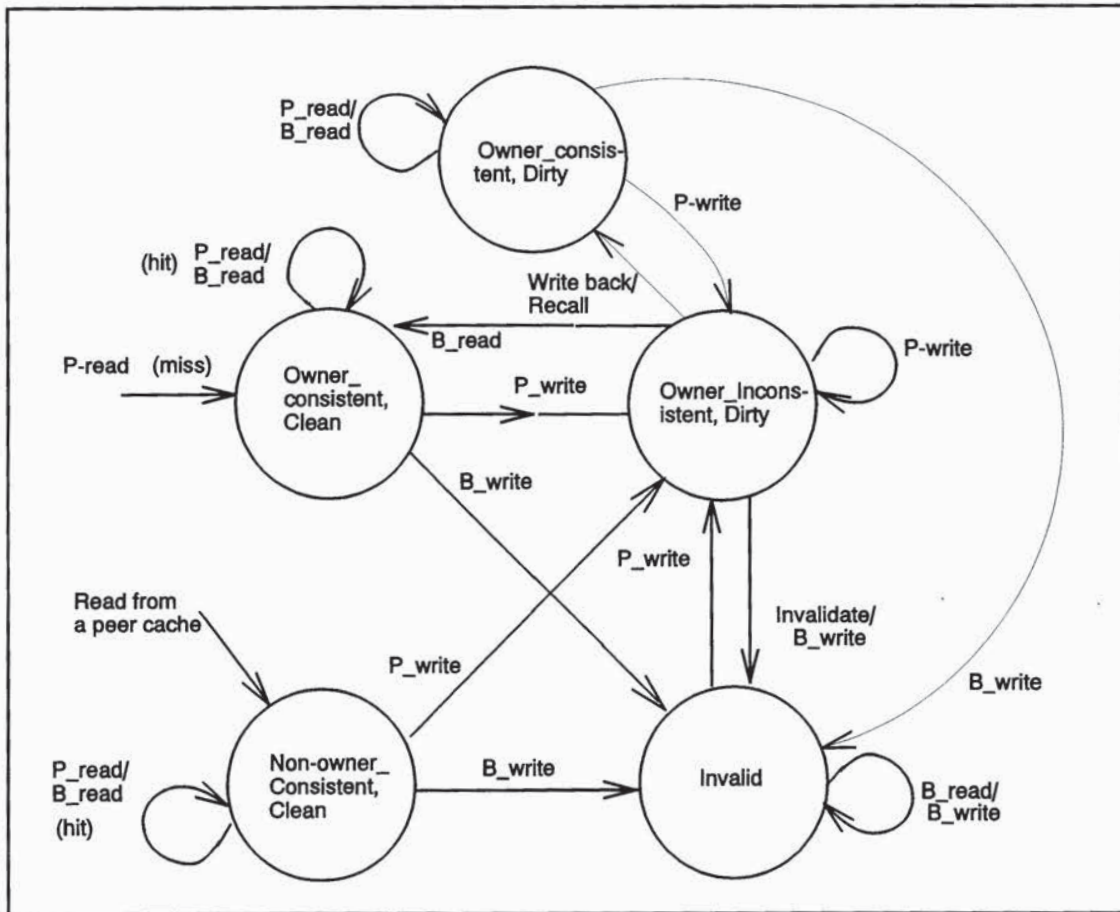


Figure 2. State diagram for a cached block at level i .

parent caches of the level 1 cache that initiated the P_read request.

Since all the copies are up-to-date with each other, the state of the block at all levels is consistent and clean. Each level has only one copy of the block and hence all the caches having this copy acquire ownership of the block. The state of the block at each level is Owner, consistent in the bus directory and Clean in the processor directory.

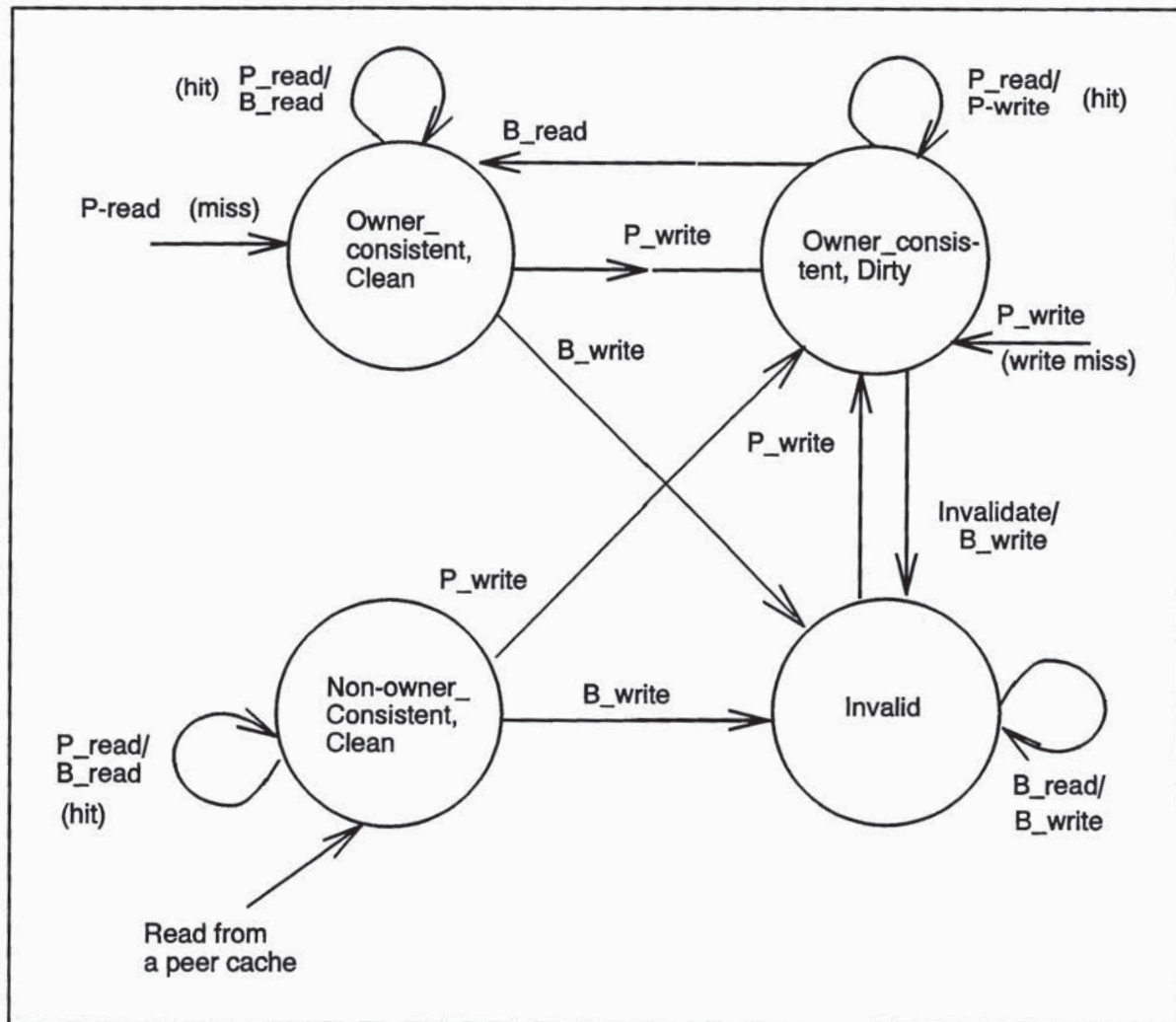


Figure 3. State diagram for a cached block at level 1.

P_read and B_read: Subsequent P_read cycles to the same block are a hit and do not alter the state of the block. In response to a bus read cycle, the owner cache supplies the block and its state remains unchanged.

P_write: A P-write request to a clean block in level 1 cache results in additional P_write and B_write requests on other levels in the system. The level 1 cache sends the P_write to its parent cache at level 2, which in turn sends the request to its parent cache at level 3 and so on. As mentioned earlier, this is necessary because the peer

caches and their descendants have to be notified that they need to invalidate their copies in response to the P_write.

A processor write request causes the state of the block to change to dirty in the processor directory at all the levels in the hierarchy. Since the block in the level 1 cache is the most updated copy, its state is Owner, consistent, dirty (Figure 3). For all other levels i ($i > 1$), the state is Owner, inconsistent, dirty (Figure 2).

B_write: A Bus write transaction on the bus forces the cache with a valid copy of the block to invalidate it.

Owner, consistent, dirty/Owner, inconsistent, dirty. The state 'Owner, consistent, dirty' is a special case of 'Owner, inconsistent, dirty.' The Owner, inconsistent state merges with the owner, consistent state at the level closest to the processor. The copy closest to the processor is always in state Owner, consistent in the bus directory. The term closest does not mean level 1 only. If a copy is absent from level 1 but does exist in only one cache at level 2, the block in level 2 cache is in state Owner, consistent, dirty. The parent caches have the copy in state Owner, inconsistent, dirty.

P_write: The first P_write request to a block causes a write miss. A write miss is made up of two parts: read miss and write hit. The block is first read into the level 1 cache and then the contents are modified in the level 1 cache.

Irrespective of the previous state of the block, a P_write request to a level one block forces its new state to be Owner, consistent, dirty as shown in Figure 3. The copies of the block in parent caches at levels higher than 1 are in state Owner, inconsistent, dirty as shown in Figure 2.

Subsequent P_writes to the same block generate a hit and the state of the block remains as Owner, consistent, dirty. Please note the state changes at levels higher than 1 are due to the P_writes and B_writes generated at those levels (as a result of P_write at level 1) and not directly due to P_write at level 1 itself.

P_read: A P_read request to a dirty and consistent block at level i is treated as a hit because it is the most up-to-date copy, being closest to the processor.

B_write: Irrespective of the previous state of the block, a B_write transaction causes all the caches with a copy (except the one that initiated the B_write) to change the state of the block to Invalid.

B_read: In response to a Bus read cycle on the level i bus, the owner cache supplies a copy of the block. If the supplying cache has the block in state 'inconsistent,' it recalls the up-to-date copy from the levels below, updates itself and then supplies to the requesting cache. During this process, the parent cache at $i+1$ updates its contents and the new state of the block in $(i+1)$ is 'Owner, consistent, dirty.'

The state changes from dirty to clean in the processor directory and Owner, consistent in the bus directory in the level i cache. The clean state is due to the fact that the parent cache at level $i+1$ updates itself when it sees the address of this block on the bus.

Special cases. Since a write back or recall is essentially a P_write operation from level $i-1$ to i , the resulting state of the block at level i is Owner, consistent, dirty. Instead of the block being in state inconsistent due to a P_write, its state is consistent because the contents are up-to-date with the one at level $(i-1)$.

Also, if an up-to-date, dirty copy has to be replaced from a level $i-1$ cache, it is written back into the parent cache at level i . The copy at level i is now up-to-date and thus is in Owner, consistent, dirty state.

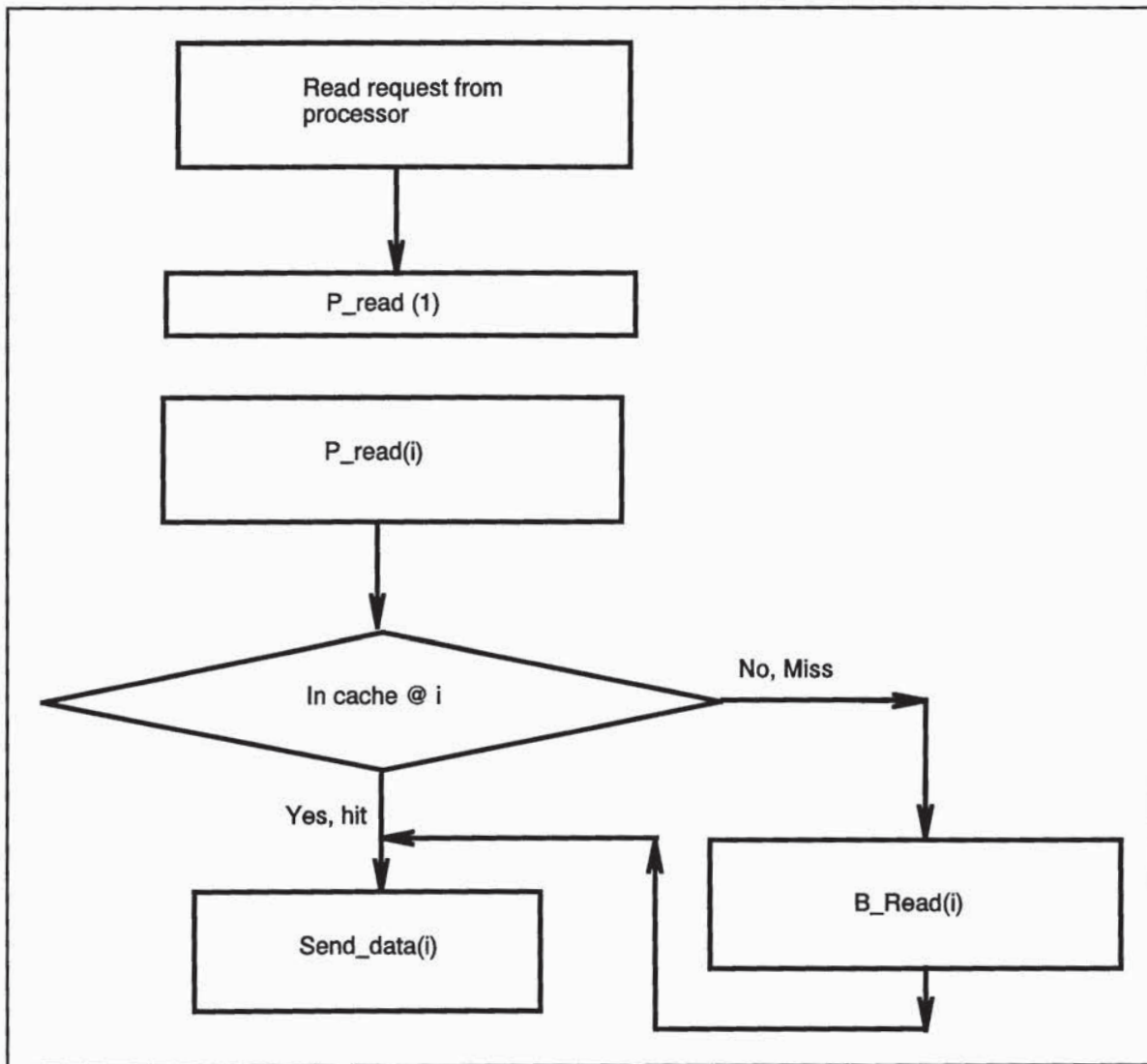
Non-owner, consistent, Clean. Let us assume that a cache connected to the (i,j) bus encounters a read miss in response to a P_read request from level $(i-1)$ and that one of the other caches connected to the (i,j) bus has a copy of the needed block. The cache puts a B_read request on the bus. In response, the peer cache which has a copy supplies it to the requesting cache. Since there can be only one owner cache for a block at level (i,j) , the block in the requesting cache is marked as Non-owner, consistent, clean.

P_read/B_read: A P_read now is a hit and no change in state takes place. The cache cannot respond to a B_read request, since it is not the owner of this block.

P_write: As mentioned above, a P_write request causes the state of the block to change to 'Owner, consistent, dirty' in the cache closest to the processor and 'Owner, inconsistent, dirty' for all other higher levels.

B_write: The cache invalidates the block in response to a B_write.

Invalid. The cache with an invalid copy does not respond to a B_read or a B_write transaction.

Flow charts for the coherence protocolFigure 4. Read request at level 1.

The symbol i refers to the level in the hierarchy which is currently being analyzed. The value of i keeps changing during the analysis, e.g., an unsuccessful B_read request at level $i=1$ is transformed into a P_read request at level $i=2$, and so on.

Overview for the read request. A processor's read request to a block appears at the level 1 cache as $P_read(1)$. If the cache has the block, it supplies it to the processor, otherwise the request is transformed to a $B_read(1)$ request and placed on the level 1 bus. An empty slot is kept ready in the requesting cache for the incoming block by $block_replacement(1)$.

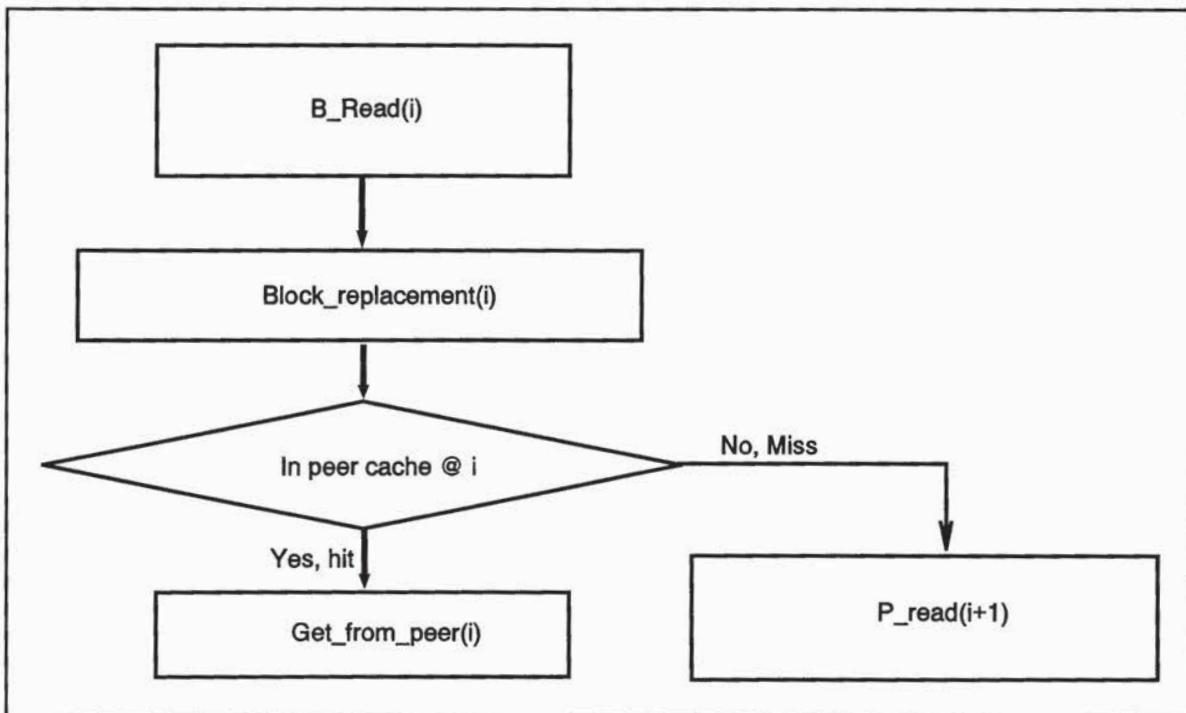


Figure 5. Bus read request at the level i bus.

If a peer cache has a copy of the block, $B_read(1)$ is a hit and the peer cache supplies the data using $get_from_peer(1)$. If the $B_read(1)$ results in a miss, i.e., none

of the peer caches at level 1 have the data, $B_read(1)$ is then transformed to $P_read(2)$ and so on until we encounter a hit with $P_read(i+1)$ or $B_read(i)$.

If the block in the peer cache is inconsistent, it has to be recalled from the levels below and then the up-to-date copy is supplied to the requesting cache.

The data is transferred from level i to level $i-1$ via $send_data(i)$ as indicated in Figure 10.

$P_read(i)$, Processor read request at level i . The read request from the processor appears as a P_read request, $P_read(1)$ at the level 1 cache. This marks the beginning of a detailed process. Please refer Figure 4.

a. Read hit. If the level 1 cache say $C1,1$ has the block in state clean or dirty, the request is treated as a hit and does not propagate any further. If the block is absent in $C1,1$, it places a B_read request on the level 1 bus, $B_read(1)$.

After it receives the block from one of the peer caches at level 1 or the processor cache at level 2, the $Send_data(1)$ module is executed which sends the block to the processor.

$B_read(i)$, Read Miss. Please refer Figure 5. A block is selected for replacement in the level i cache. If one of the peer caches has a copy, $B_read(i)$ is a hit and the copy is supplied using $get_from_peer(i)$. An unsuccessful $B_read(i)$ request leads to a $P_read(i+1)$, i.e., a request to the next higher level cache. This operation is recursive until a hit is encountered.

$Get_from_peer(i)$. Please refer Figure 6. The state of the block in the peer cache is checked to see if it is consistent. If it is an updated copy, the data is supplied to the

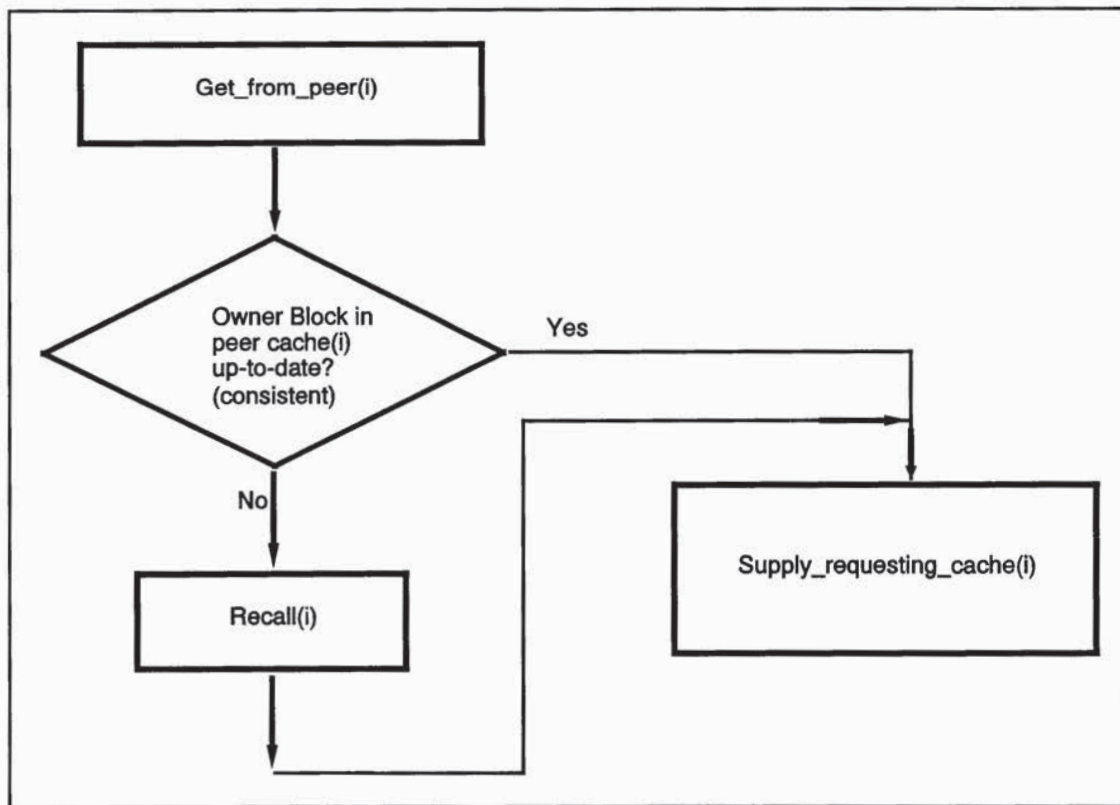


Figure 6. Receive data from peer cache at level i .

requesting cache by *Supply_requesting_cache(i)*, shown in Figure 7. The state of the block in the requesting cache is marked as Non-owner, consistent, clean.

The parent cache at $(i+1)$ also updates its copy and the block in the parent cache is in state Owner, consistent, dirty. The state is dirty because as explained earlier, the update is nothing but a P_write operation. Also, after the update, the copy at $(i+1)$ is updated but not at $(i+2)$. If the Owner peer cache has an inconsistent copy, it initiates the Recall process.

Recall(i). The Owner peer cache at level i needs to fetch a consistent copy of the block from levels below. The details are shown in Figure 8.

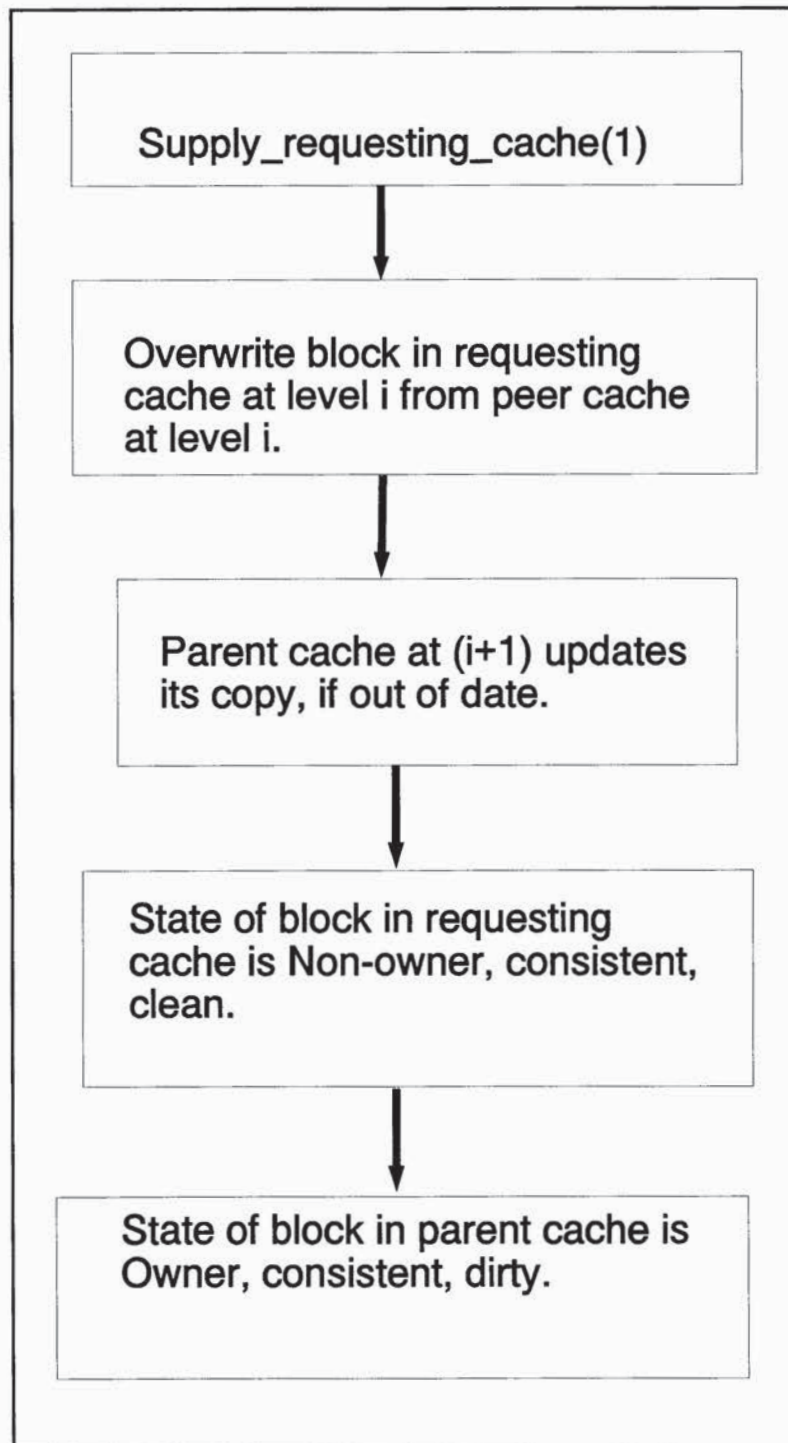


Figure 7. Supply data to the requesting cache.

Note: We are looking for a consistent copy of the block (and not a *dirty* copy) from the levels below. This is because a dirty block may be inconsistent, i.e., not up-to-date.

The cache at level i checks to see if the copy at level $(i-1)$ is consistent. If yes, the contents at level i are updated by the descendant cache and the up-to-date copy is supplied to the requesting cache.

If no, the recall request is placed on the next lower level bus, i.e., $(i-2)$ and the entire process is repeated again until we reach the cache with a consistent copy.

The recall process updates the contents of all the parent caches in the intermediate levels. The states of the blocks in all the caches that were a part of the recall process is Owner, consistent, clean.

For example, assume $C_{3,1}$ had placed a $B_read(3)$ on the level 3 bus. $C_{3,2}$ has a copy but is in state inconsistent. The consistent copy of the block resides in $C_{1,7}$. The operation is explained as follows:

The copy at level $(2,2)$ bus is checked for consistency. It turns out to be inconsistent, so the request is placed on the level $(1,3)$ bus. $C_{1,7}$ responds and updates its parent cache, $C_{2,3}$. The states of the block in $C_{1,7}$ and $C_{2,3}$ are Owner, consistent, clean and Owner, consistent, dirty respectively. Once $C_{2,3}$ gets updated, it updates $C_{3,2}$. The states of the block in $C_{2,3}$ and $C_{3,2}$ are Owner, consistent, clean and Owner, consistent, dirty. $C_{3,2}$ now has a consistent copy of the block and the recall operation is completed.

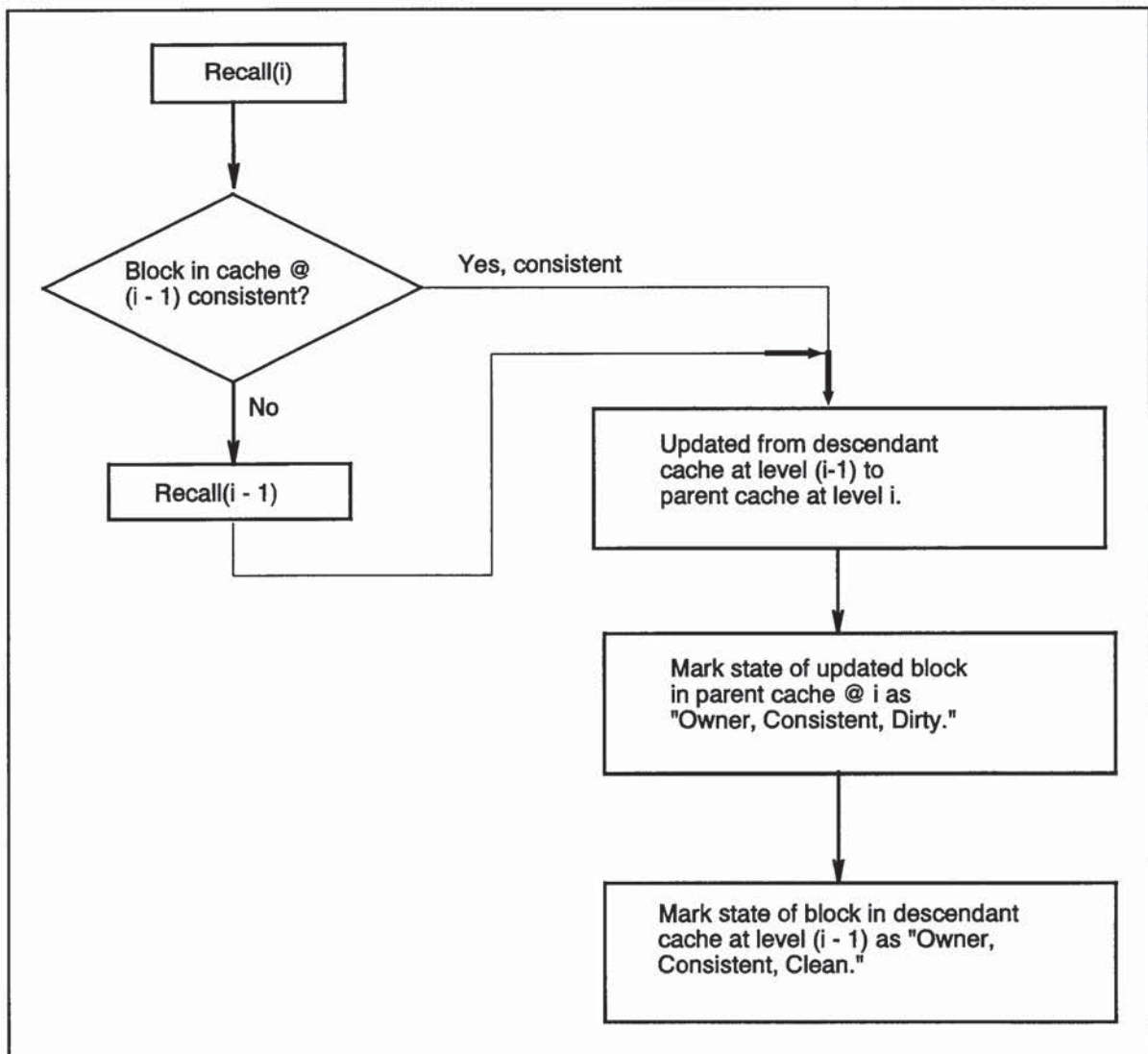


Figure 8. Recall(i).

Block_replacement(i). The block replacement algorithm in the case of a multi-level hierarchical architecture is complicated as compared to a single level system. The primary reason for the added complexity is the Multi Inclusion principle (MLI). Please refer to the flow chart in Figure 9.

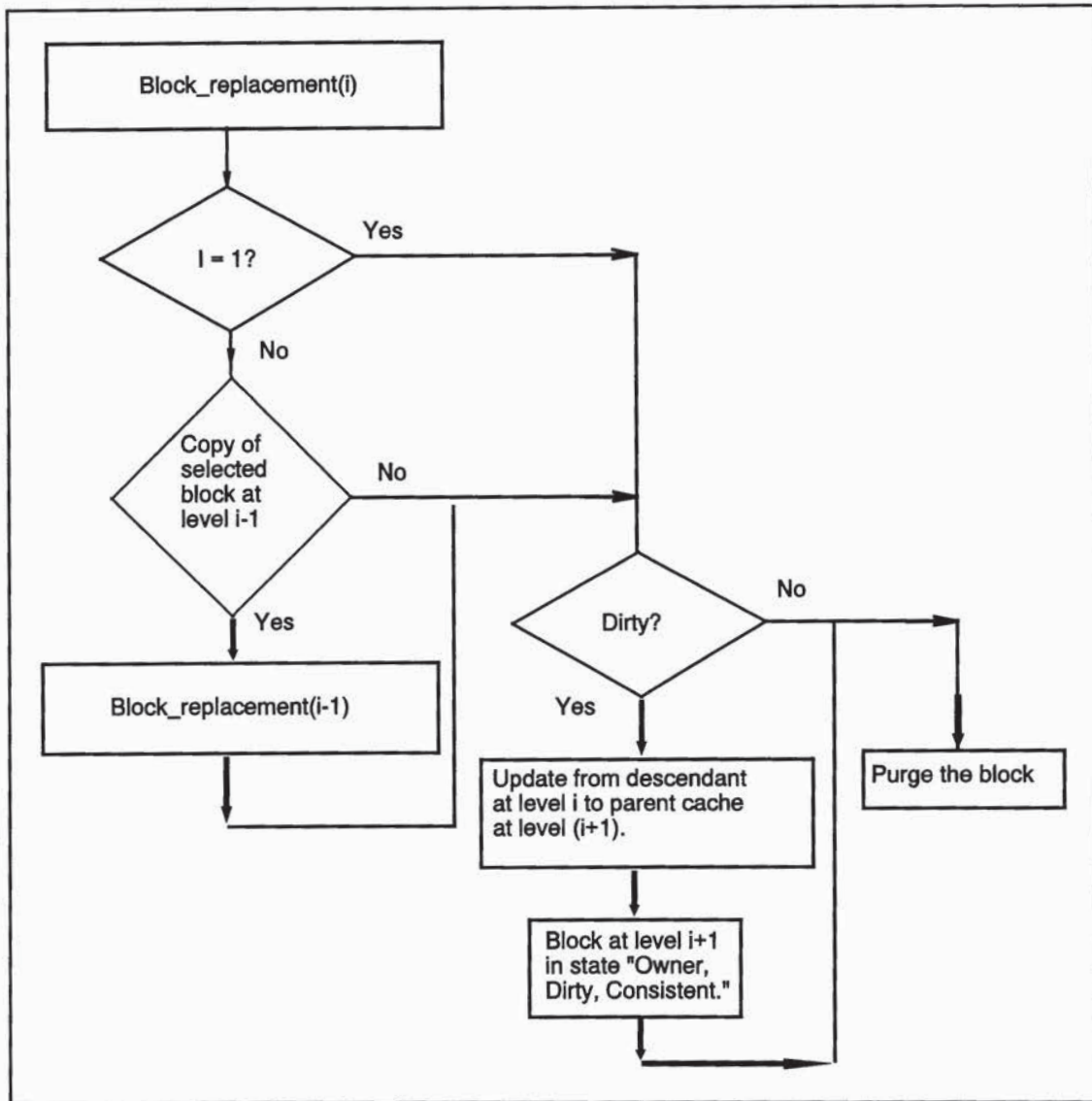


Figure 9. Block replacement at level i .

Before a block can be purged from level i , all copies of this block in levels lower than i need to be purged. This is done to maintain MLI.

If the copies are in state clean, then the replacement process goes down until it finds the lowest level cache with a copy of the block. All the copies in this path get

purged.

However, if the copies at lower levels are in state dirty, then starting from the cache with a copy at the lowest level, we need to update their parent caches before purging the block.

For example, a dirty block in a level 3 cache needs to be purged and there are dirty copies in descendant caches at levels 2 and 1.

The replacement request comes all the way down until level 1, updates the parent cache at level 2 and then purges the dirty block in the level 1 cache. The state of the block in parent cache at level 2 is Owner, consistent, dirty. Next, we update the parent cache at level 3 and then purge the block at level 2. Now, the state of the block in the level 3 cache is Owner, consistent, dirty.

Finally, the level 3 cache updates the main memory and then the block at level 3 is purged. Thus we ensure the consistency of data between main memory and rest of the system.

Send_data(i). The *send_data(i)* module transfers data from a parent cache at level i to the descendant cache at level $(i-1)$. Since the descendant cache will be the only one to have a copy of this clean and consistent block at level $(i-1)$, the state of the block is Owner, consistent, Clean.

Once the data has reached level 1, the level 1 cache supplies the data to the processor.

Overview for the write requests. A P_write request to a clean block necessitates the need to invalidate the copies of this block in the peer caches and their descendants

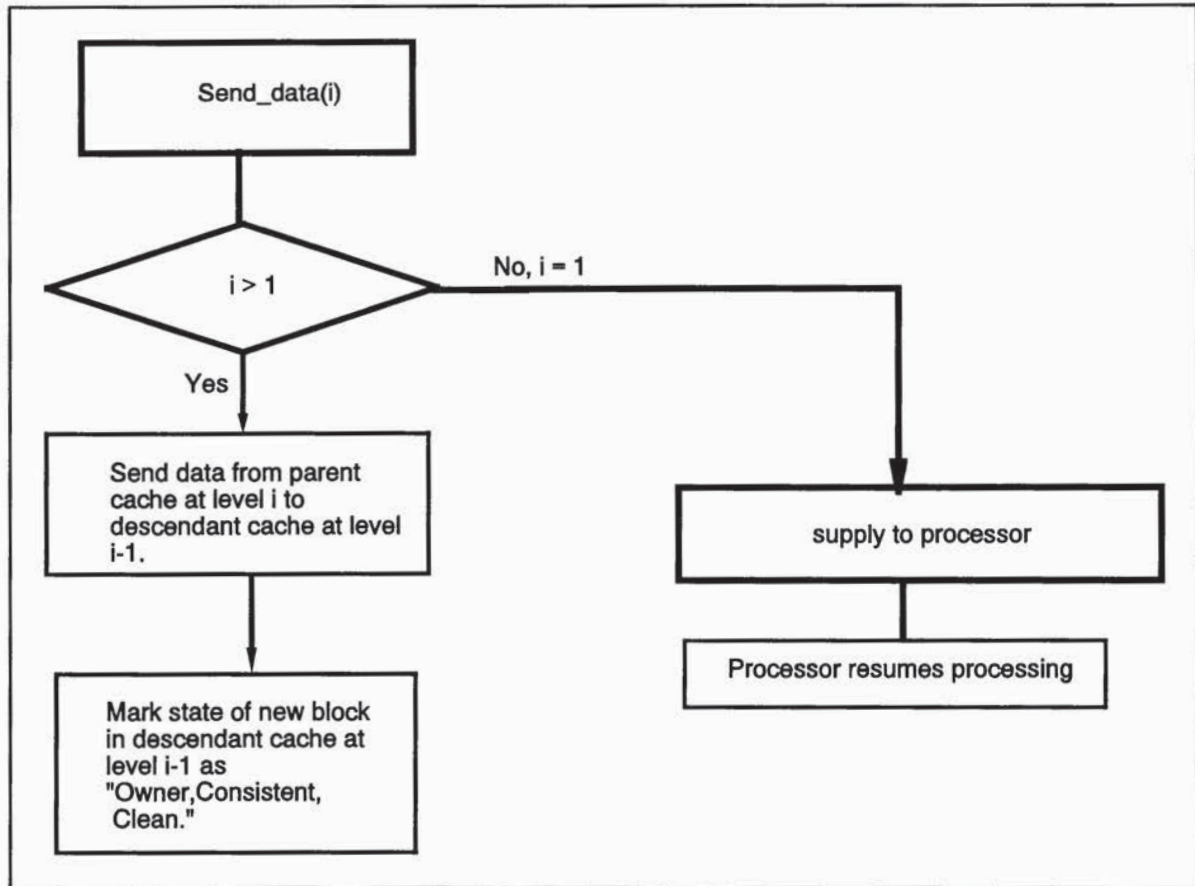


Figure 10. Send data to the level below.

in the system. This is done to ensure coherence of data across the entire system.

The invalidate request is placed on the level 1 bus. Any peer cache that has a copy invalidates its copy. Invalidate is nothing else but a B_write transaction on the system bus. The request is now put on the next higher level bus and the invalidation process is repeated. This time however, all the peer caches that respond to the B_write also check their descendant caches for a copy of the block.

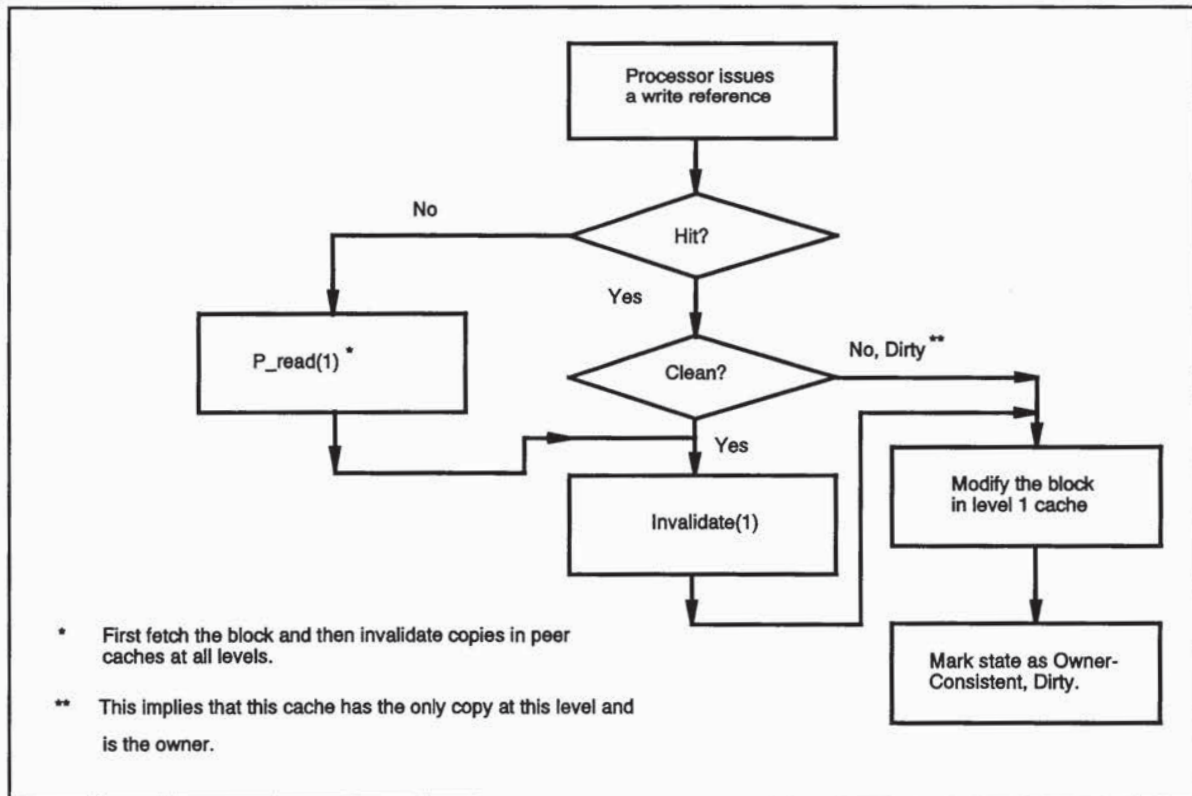


Figure 11. Overview of a P_write request

If any descendant has a copy, the invalidation request is placed on the level (i-1) bus and all caches connected to this bus invalidate their copies. The invalidation request keeps going down until all the caches in that path have invalidated their copies.

The invalidation request keeps going up until we reach the highest level or it encounters a cache with the block in state dirty. If the block is in state dirty at level i, it means that it is the only copy at level i and there is only one copy at each of the levels above; so there is no point in sending invalidation request any higher.

At this point, an Invalidation acknowledge signal is sent from the highest level to the next lower level. The acknowledge signal finally reaches the processor. The processor can go ahead and update the block.

A write miss is a read miss operation followed by a write hit. The steps involved in a 'read miss' operation are as covered in sub-section titled P_read(i).

Processor read request at level i. The write hit operation is the same as explained above.

a. Write hits. If the processor's write request is a hit in the level 1 cache say C1,1, and the block is in state dirty, it is updated. The state of the block remains as Owner, consistent, dirty. Please refer Figure 11. Please remember that the state of the same block in its parent caches is Owner, inconsistent, dirty.

If the state of the block is clean, then copies in peer caches at other levels in the hierarchy need to be invalidated. This is done by the Invalidate(i) module, described in the Figure 12. There can be only one copy of this block at each level in the hierarchy and this copy can only be present in the parent caches of C1,1.

Once all the required copies have been invalidated, the processor updates the block in C1,1. The state of the block will be Owner, consistent, Dirty.

Invalidate(i). Please refer to Figure 12. The invalidate process begins by placing an invalid request on the level 1 bus. Any peer cache that has a copy of this block invalidates it. Before the invalidate request moves up the hierarchy, two checks need to be performed:

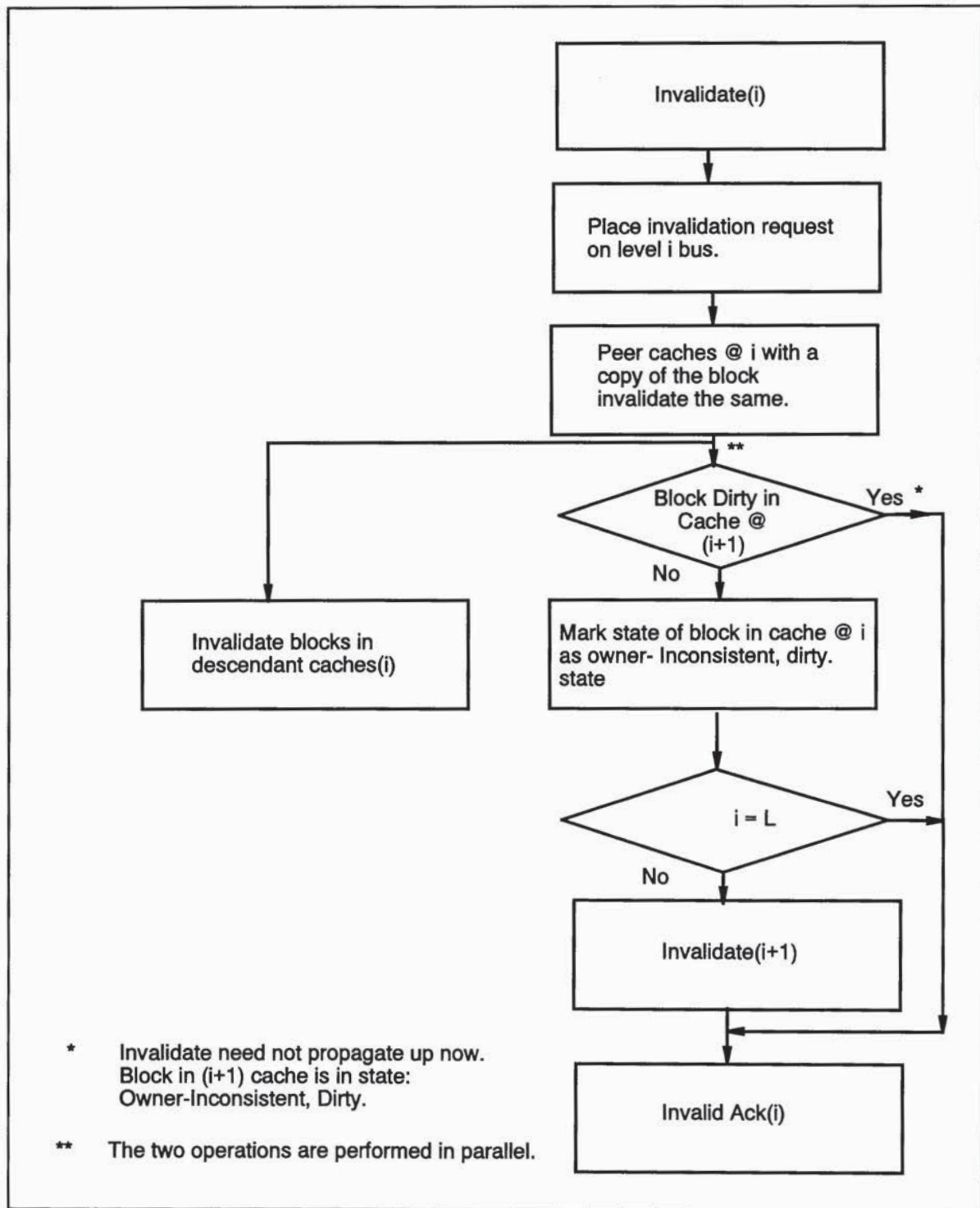


Figure 12. Invalidate process at level i .

Check 1: Does the parent cache at level $i+1$ have the block in state dirty?

If a block in a level $i+1$ cache is in state dirty, then all the parent caches at levels $i+2$ through L must have the block in state dirty. For example, if the block in a level 1 cache is dirty, then all the parent caches have the block in state dirty as well.

Check 2: Are we at the top of the memory hierarchy?

If we have not reached the top and the block in the parent cache at level $(i+1)$ is not in state dirty, we can send the invalidate request to the next higher level $(i+1)$. If the invalidation request need not propagate any further, the owner cache at level i sends an Invalid acknowledge signal to its descendant cache at level $i-1$. This is taken care of by the Invalid Ack(i) module as explained later.

Any peer cache at level i that had a copy of this block also checks the contents of its descendant caches at level $(i-1)$. If they have a copy of the same block, it is invalidated. This is a recursive process and is explained in the 'invalidate blocks in descendant caches(i)' module.

The two operations mentioned above go on in parallel, i.e., while one invalidate request is going up the hierarchy, other requests are also going down.

Invalid ack(i). The owner cache at level i sends the invalid acknowledge signal to its descendant cache at level $i-1$. This happens only if i is greater than 1.

The processor writes to the block in question only after the invalidation acknowledge signal has reached level 1 cache.

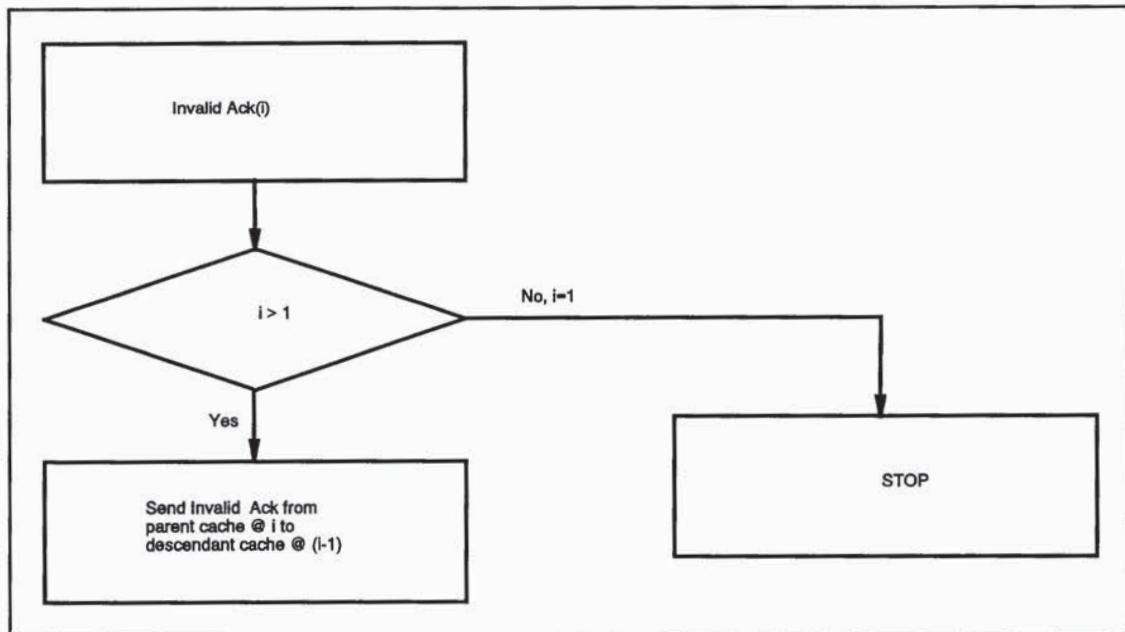


Figure 13. Sending Invalid acknowledge signal to level below.

Invalidate blocks in descendant caches(i). All the peer caches at level i ($i > 1$) that responded to the invalidate request also check to see if their descendant caches at level $(i-1)$ have a copy of the block.

Note: The flow chart conveys the impression that only one peer cache at a time is invalidating the contents of its descendant caches. This is not true and all the peer caches are invalidating their descendants in parallel.

If the answer is yes, then the invalidate request is placed on the each of those buses at level $(i-1)$. This is again a recursive process in that all the level $(i-1)$ caches that respond now check the contents of their descendant caches at level $(i-2)$ and any copies found are invalidated. This process is repeated until we reach level 1 or a level below which there are no copies.

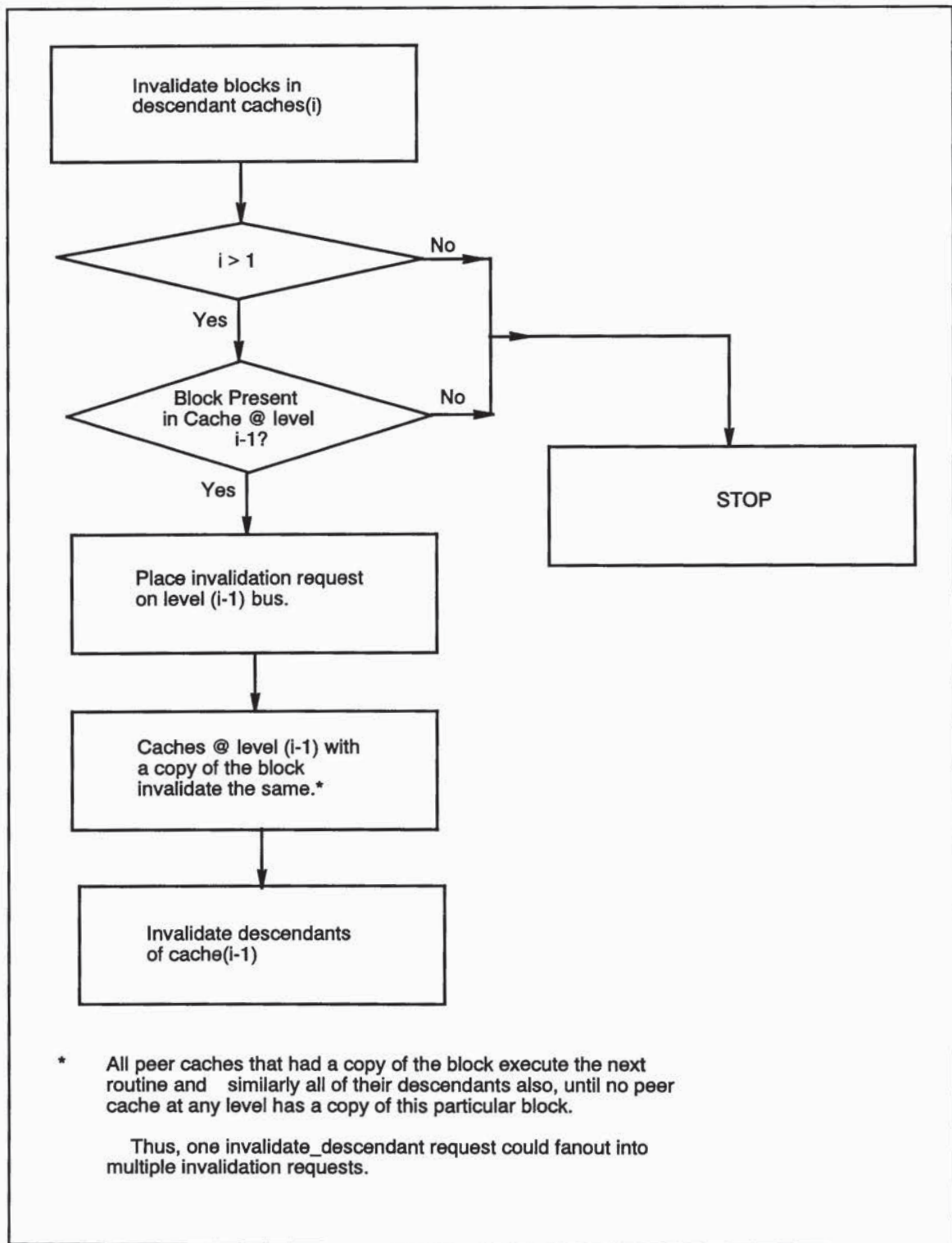


Figure 14. Invalidating blocks in descendant caches.

At this stage, there is only one copy of the block at each level in the hierarchy. Moreover, this copy exists in only the parents of C1,1 cache and not in the peer caches.

Write misses. Please refer to Figure 11. A write miss as mentioned earlier is a read_mod transaction. This means the block is first read into the level 1 cache and then modified. The complete transaction is composed of two parts: read miss and write hit to a clean block.

Hence, first the P_read(1) module is executed which brings the block into the level 1 cache. Next the invalidate(1) function is executed, which modifies the states of the block as dirty in the parents of the level 1 cache. After this is done, the processor goes and modifies the contents of the block and the state changes from "Owner, consistent, clean" to "Owner, consistent, dirty." This makes write miss as the most expensive operation to perform.

Summary

This chapter covered the salient features of the TREEBUS architecture including the coherence protocol in great detail and a sincere and dedicated effort has been made to present the details in a form that are easy to read and understand.

The later half of the chapter laid the foundation for the development of the mathematical model that will be covered in the next chapter.

CHAPTER IV

MODEL DEVELOPMENT

OVERVIEW

The last chapter explained the TREEBUS architecture and the cache coherence protocol in great detail. This chapter will focus on the development of a mathematical model for its behavior. The structure of the model will be identical to that of the flow charts covered in Chapter 3.

This chapter is divided into four sections; *Read accesses* are modelled in section 1, *Write accesses* in section 2, the third section focusses on the development of a *cost model* and finally, we will cover the transformation of *higher level* input parameters, e.g., degree of sharing of data between different clusters, etc., that relate more directly to application program characteristics [5] into *low level* parameters, e.g., hit ratios at each level, etc. The model is implemented in 'C' and uses low level parameters as inputs.

MODEL FOR READ ACCESSES

The file *read.c* contains the program that models the read accesses. Please refer to Appendix I for a detailed listing. The program consists of a number of functions each of which return the time taken to accomplish a certain task.

Functions in Read.c

P_read(i). This function returns the time it takes to complete a read access at level i. P_read(1) returns the time taken to complete a read access initiated by the processor.

B_read(i). If the cache that encountered the processor's read access at level i suffers a miss, this function returns the time it takes to fetch a copy from one of the peer caches at level i or from the caches at levels above.

Send_data(i). The time taken to transfer a block of data from the level i cache to the level (i-1) cache.

Block_replacement(i). The time it takes to create space in a cache at level i for an incoming fresh block. If the block to be replaced is dirty, it has to be written back to the parent cache. Also, if there are copies of this block in levels below, they need to be invalidated to maintain multi level exclusion (MLI)

Get_from_peer(i). The time it takes to get a copy of the block from a peer cache at level i.

Recall(i). If the peer cache at level i has an out-of-date (inconsistent) copy of the block, Recall(i) returns the time taken to update this copy from its children in levels below.

Supply_requesting_cache(i). The time taken to supply a copy from the peer cache to the requesting cache at level i.

These functions were explained in detail in chapter 3 using flow charts. The variables used by these functions are explained below:

Variables used in Read.c

LIMIT: The maximum number of levels in the hierarchy, L , plus 1, i.e., $LIMIT = (L+1)$.

Hit_cache[i]: The hit ratio for any cache at level i . It is assumed that the hit ratio is the same for all caches at level i that are trying to satisfy a processor request.

Hit_cache[LIMIT] is assumed to be 1, which means that we will always find the block in the main memory.

Hit_peer[i]: The hit ratio for any of the peer caches at level i . This value is assumed to be the same for peer caches connected to any bus j , at level i . Hence, ' j ' is not included in the description.

Peer_consistent[i]: Given that the peer cache has a copy of the block in question, the probability that it is also up-to-date, i.e., consistent. Peer_consistent[1] is always 1, which means that the data in any level one cache is always consistent.

t_read_access: The mean time to complete a read request initiated by the processor.

UPDATE_PARENT_CACHE: The time taken to update a block in the parent cache at level i using the data in the descendant cache at level $(i-1)$.

P_read: The probability that a given access is a read access. The probability of a write access is $(1 - P_{\text{read}})$ and is denoted as P_write.

Timing equations

The output of the program Read.c is the average time taken to complete a processor read access. This process is recursive in nature (as shown in the flow charts in the last chapter) and the program is written to take advantage of this feature.

The timing equations are as follows:

$$\begin{aligned} t_{average_access_time} &= t_{read_access} + t_{write_access} \\ t_{read_access} &= p_{read} \times P_{read}(1) \end{aligned} \quad (5)$$

$$\begin{aligned} P_{read}(1) &= (Hit_cache[1] \times send_data(1)) + \\ &\quad ((1 - Hit_cache[1]) \times (B_{read}(1) + \\ &\quad send_data(1))) \end{aligned} \quad (6)$$

If the cache that encountered the processor's read request at level 1 cannot service a processor read request, B_read(1) is called, i.e., an effort is made to satisfy the request at the same level by one of the peer caches.

Simplification of equation (6) leads to equation (7).

$$P_read(1) = ((1 - Hit_cache[1]) \times B_read(1)) + Send_data(1) \quad (7)$$

The bus needs to be acquired before any operation can begin. Also, space is created in the cache for the incoming block.

If $B_read(1)$ is also unsuccessful, i.e., none of the peer caches at level 1 have a copy of the block, $P_read(2)$ is called as shown in equation (8). $P_read(2)$ will in turn call $B_read(2)$ and so on. This operation continues recursively till a hit is encountered in one of the higher level caches or the request reaches the main memory.

If one of the peer caches at level 1 has a copy, $B_read(1)$ is a hit and the peer cache supplies the data to the requesting cache.

$$B_read(1) = (Hit_peer[1] \times Get_from_peer(1)) + (P_read(2) \times (1 - Hit_peer[1])) + (Bus_access_time(1) + Block_replacement(1)) \quad (8)$$

If none of the caches in the system have a copy then this process stops only when i equals L i.e., the processor's read request is finally serviced by the main memory. An assumption is made here that main memory always has a copy of the needed block, i.e., $Hit_cache[LIMIT]$ is always 1.

The time taken for a read access at any level i , where $i > 1$, is given by $P_read(i)$ as shown below:

$$P_read(i) = ((1 - Hit_cache[i]) \times (B_read(i) + Bus_access_time(i-1))) + send_data(i) \quad (9)$$

Note: An additional term is included in the equation for $P_read(i)$ (equation 9), where i is greater than 1. The extra term is for the additional bus access needed when the data block is sent from the higher levels of the hierarchy to the lower levels. Equation 8 does not have this extra term for $bus_access_time()$, because this is needed only if there is a miss in all the caches connected to the level 1 bus.

The time taken to satisfy a read request on the level i bus, where $i > 1$, is given by $B_read(i)$:

$$B_read(i) = (Hit_peer[i] \times Get_from_peer(i)) + (P_read(i+1) \times (1 - Hit_peer[i])) + (Bus_access_time(i) + Block_replacement(i)) \quad (10)$$

As mentioned earlier, if $B_read(i)$ is successful, a copy is fetched from one of the peer caches at level i , (equation 11). This copy in the peer cache may be

inconsistent. In that case, it needs to be updated by recalling the most up-to-date copy from the descendants below. Please refer equation 12 for the Recall time calculations.

$$\begin{aligned}
 \text{Get_from_peer}(i) = & \text{Supply_requesting_cache}(i) + \\
 & ((1 - \text{Peer_consistent}[i]) \times \\
 & \text{Recall}(i))
 \end{aligned}
 \tag{11}$$

$$\begin{aligned}
 \text{Recall}(i) = & (\text{UPDATE_PARENT_CACHE} + \text{Bus_access_time}(i-1)) + \\
 & ((1 - \text{Peer_consistent}[i-1]) \times \text{Recall}(i-1)) + \\
 & \text{Bus_access_time}(i)
 \end{aligned}
 \tag{12}$$

Recall is a recursive process and it keeps going down in the hierarchy until an updated block is found in one of the descendant caches or recall reaches level 2. Please note that Recall(1) is 0, i.e., the copy at level 1 is always consistent and there is no need to perform any recall operation. The recall operation also suffers an additional bus access cycle during the update process. It has to first go down a level to get an updated copy of the block and then come back to the original level and access the bus again to supply the block to the requesting cache.

MODEL FOR WRITE ACCESSES

The program write.c contains the program that models the write accesses. Please refer to Appendix II for a detailed listing. Each of the functions returns the time taken to accomplish a certain task. The program consists of the following functions as explained below.

Functions in write.c

P_write(1). The time taken to finish a write access issued by the processor.

Invalidate(i). The amount of time it takes to invalidate the copies of a block in the peer caches at level i through L.

P_read(1). In the case of a write miss, we have to first fetch the copy of the block before proceeding with the write. Hence the need to call P_read(1).

Invalid_ack(i). The time it takes to send an Invalid acknowledge signal from a level i cache to its descendant at level (i-1).

Variables used

The variables used in the functions are as follows:

P_clean[i]: The probability that the block in the level i cache is in state clean. The probability that the block is in state dirty is given by $(1 - P_clean[i])$ and is denoted by P_dirty[i].

Note: If $P_clean[i] = 0$, then $P_clean[i+1]$ through $P_clean[L]$ is also equal to 0. This means that if the block is in state dirty at level i, all the parent caches also must have the block in state dirty.

$t_write[i]$: The write access time for the memory at level(i). $t_write[i] \leq t_write[i+1]$, i.e., the memory chips at higher levels are generally slower in speed when compared to those at lower levels.

$t_invalidate$: This is the time taken to drive the invalidation request on to the bus. The time is typically equal to one bus cycle.

Timing equations for $P_WRITE(1)$

In the case of a write access issued by the processor, $P_write(1)$, there are three possibilities; a write miss is encountered, a hit is encountered but the block is in state clean and finally the processor's access encounters a hit to a dirty block. Please refer Figure 3.11 in chapter 3.

The mean time to complete a write access is denoted by t_write_access and is calculated as shown below:

$$t_write_access = p_write \times P_write(1) \quad (13)$$

The equation for $P_write(1)$ is as follows:

$$P_write(1) = t_write_hit_clean + t_write_hit_dirty + t_write_miss \quad (14)$$

Let us now look at each of the three parts that make up $P_write(1)$:

$t_write_hit_clean$. It is the mean time to complete a write access, if the block was present in state clean in the level one cache (equation 15). All the copies of this block that are present in the peer caches and their descendants need to be invalidated before the write can go through.

$$t_write_hit_clean = (Hit_cache[1] \times p_clean[1]) \times (Invalidate(1) + t_write[1]) \quad (15)$$

The mean time to complete a write access, if the block was present in state dirty in the level one cache, is represented by $t_write_hit_dirty$ (equation 16). If the block is dirty, then the processor can go ahead and update the contents of the block without notifying any other cache.

$$t_write_hit_dirty = (Hit_cache[1] \times (1 - p_clean[1])) \times t_write[1] \quad (16)$$

If the processor encounters a miss, then a Read_mod [5] cycle is initiated. The block is first brought into the cache, then all the copies in peer caches are invalidated as shown in equation 16 and finally the block is modified. The time taken for this process is denoted by t_write_miss (equation 17).

$$\begin{aligned}
 t_{write_miss} = & (1 - Hit_cache[1]) \times \\
 & (P_read(1) + Invalidate(1) + \\
 & t_{write}[1])
 \end{aligned}
 \tag{17}$$

Invalidate(1). Please refer equation 18. The bus is acquired and the invalidation request is placed on the bus. If the block in question is in state clean at level 2, the invalidate request is sent upward so that all the peer caches at level 2 can invalidate their copies of this block. Also, an Invalid acknowledge signal has to be sent to the level 1 cache from the level 2 cache, which requires another bus access. Equation 19 deals with the general case.

$$\begin{aligned}
 Invalidate(1) = & bus_access_time(1) + t_invalidate + \\
 & ((Bus_access_time(1) + Invalidate(2)) \times \\
 & p_clean[2]) + Invalid_ack(1)
 \end{aligned}
 \tag{18}$$

Note 1: Invalid_ack(1) returns a value of zero. This is because no acknowledge is sent from the level one cache to the processor.

The term 'Y[i]' in equation (19) helps us to take care of the limiting situation when the invalidate process is at the top of the hierarchy, e.g., in a 3 level system, L = 3, and Invalidate(3) gives us:

$$bus_access_times(3) + t_invalidate + Invalid_ack(3)$$

$$\begin{aligned}
\text{Invalidate}(i) = & \text{bus_access_time}(i) + t_invalidate + \\
& (p_clean[i+1] \times Y[i] \times \\
& (\text{Bus_access_time}(i) + \text{Invalidate}(i+1))) + \\
& \text{Invalid_ack}(i)
\end{aligned}
\tag{19}$$

as $Y = 0$. The invalidation process need not go any higher and also there is no need for the extra bus access cycles when we are at the highest level.

$$Y[i] = \begin{cases} 1, & 1 \leq i \leq (L-1) \\ 0, & \text{otherwise} \end{cases}
\tag{20}$$

Note 2: Invalidate(LIMIT) returns a value of zero. The invalid request only goes till the highest level of the hierarchy, i.e., L.

COST MODEL

The cost model was discussed earlier in Chapter 2. This is a preliminary model and it is incorporated to give a balanced view about the TREEBUS architecture. This architecture promises a high level of performance but the total system memory size and cost have the potential to explode for even moderately small systems employing about 256 - 1024 processors. The earlier research work done on this topic [6, 13, 25] does not address this important topic at all.

Wilson [13] has suggested that the size of the parent cache at level i should be an order of magnitude larger than the sum of its descendants at level $i-1$. Thus the total memory size is:

$$Totalsize = (\sum_{i=1}^{i=L} (N_i \times n_{i,1}) \times C_{size,i}) + C_{Size,L+1} \quad (21)$$

where $C_{size,i}$ is given by:

$$C_{Size,i+1} = (n_{i,1} \times C_{Size,i}) \times \alpha \quad (22)$$

The product $(N_i \times n_{i,1})$ in equation (21) gives the total number of caches at a particular level i (number of buses $\times$ branching factor). The term α in equation (22) is referred to as the MLI factor and $1 \leq \alpha \leq M$, where M is a reasonable size number. The higher the value of α , the higher is the value of hit ratio $hit_cache[i]$ which reduces the bus traffic at all levels. But, the cost of the memory sub-system increases very rapidly as α increases even by a small amount. When $\alpha=1$, the size of the parent cache at level i is equal to the sum of the size of its children caches at level $i-1$. This is done to enforce Multi level Inclusion principle in the memory hierarchy.

SRAMs are generally an order of magnitude more expensive than DRAMs (estimate based on preliminary cost comparison efforts).¹ So, if the DRAM costs \$ x /byte, SRAM will cost \$ $10x$ /byte. The model assumes that all the memories at level 2 through the highest level are of the same speed and hence cost the same.

The total cost of the memory system is shown in equation (23).

$$\begin{aligned}
 Totalcost = & \left(\sum_{i=2}^{i=L} (N_i \times n_{i,1} \times C_{size,i} \times x) + \right. \\
 & (C_{Size,L+1} \times x) + \\
 & \left. (C_{size,1} \times N_1 \times n_{1,1} \times 10x) \right)
 \end{aligned}
 \tag{23}$$

MAPPING OF HIGH LEVEL PARAMETERS TO LOW LEVEL PARAMETERS

The model discussed in this chapter is based on low level parameters, e.g., `hit_cache[i]`, `p_clean[i]`, and so on. It is extremely difficult to find an accurate value for these parameters, because they are totally dependent on the characteristics of the program or application being run on the system. It is difficult to measure these parameters because there are no commercial systems designed around the TREEBUS architecture and hence there are no program traces available.

¹We called a number of dealers and distributors for pricing information but they were totally uncooperative. The first question I would be asked was "Which company do you work for?" I would reply that I'm a graduate student at Portland State University. They would not seem impressed with my reply because probably I was not going to place any orders with them.

Our strategy is to sweep some of the parameters, i.e., during final analysis, the variable of interest is allowed to take values over a wide range, part of which may not even be realistic. This is done so that the designer can have a good understanding of the impact of the variable on system performance.

Mapping

An analytical model is discussed in [5]. Some examples of higher level parameter are:

1. fraction of memory references to shared data blocks that reside within a cluster sub-rooted at a particular level.
2. fraction of memory references to a cluster rooted at one level, say k , that miss in a cache because they have been invalidated by the 'writes' from another processor. Note that if another processor is modifying a shared block, all the peer caches that have a copy of the block need to invalidate their copies.

This fraction would indicate the frequency with which other processors write to a particular shared block, as a result of which a particular processor finds the block invalidated frequently.

3. fraction of processors that read the block after it has been modified by some other processor. This high level parameter has a direct impact on the value of p_{clean} . A bus read to a block in state dirty changes its state to clean in both the requesting and supplying cache.

If this fraction is a large, the block stays in state clean for most of the time thereby increasing the time to complete a write request. This is because the invalidate

operation assumes that if one of the caches at level i has the block in state clean there is a possibility that one or more caches also have a copy which need to be invalidated.

The mapping of higher level parameters to the low level parameters is left up to the designer of the memory system and a user of this model. The mapping process and even the selection of high level model parameters is an actively pursued research problem.

CHAPTER V

RESULTS AND FINAL ANALYSIS

OVERVIEW

This chapter uses the mathematical model developed in the last chapter to analyze the architecture in greater detail. There are a number of parameters that can potentially affect system performance. This makes it hard to analyze the architecture. Also, since the same parameter at different levels in the hierarchy can have different values, we need to consider that aspect as well during the analysis.

For example, the probability of finding a block in the peer caches at various levels is different and depends on the data sharing and access patterns of the program or application being used. During our analysis, we will vary two parameters at a time and study their impact on the system performance.

The subject of focus is the average access time which includes both the read and write accesses. An attempt is made to understand the impact of each parameter on the average access time, which is the key measure of memory system performance. Since this value is made up of many different processes in the system, e.g., invalidate, bus access, block replacement, etc., we also analyze the effect of some of these parameters on a particular process whenever it was felt necessary to better illustrate a point.

In an attempt to generate realistic numbers for the analysis, we referred to [5], [25] and [26].

PERFORMANCE ANALYSIS

The values used as inputs to the model are listed in Tables V and VI below. The values are for a four level TREEBUS hierarchy ($L=4$). Table V lists the values of hit ratios in the caches, hit ratios in the peer caches, and the probability that the blocks are in state clean and consistent at various levels.

TABLE V
INPUT VALUES TO THE MODEL: SET I

level i	Hit_ Cache[i]	Hit_ peer(i)	peer_ consistent[i]	p_read	p_write	p_clean[i]
1	0.910	0.800	1.000	0.8	0.2	0.300
2	0.950	0.750	0.850	0.8	0.2	0.600
3	0.990	0.750	0.850	0.8	0.2	0.600
4	1.000	0.750	0.850	0.8	0.2	0.600

The values of hit_peer[i], p_clean[i] and peer_consistent[i] for $i > 1$ are the same in table V. This is done just for reasons of simplification. During the analysis stage, the values for these parameters are varied over a wide range and the effect observed.

The times in Table VI are given in terms of bus cycles. The system bus in the Sequent symmetry system operates on a 10 Mhz clock, i.e., one bus cycle = 100 ns. The Symmetry is designed around an Intel 80386 processor running at 16 Mhz, (processor cycle time = 62.5 ns). The system bus employs a 64 bit wide data bus.

During burst mode of operation, a 16 byte transfer should not take more than 3 bus cycles (assuming no wait states).

Also, during the sensitivity analysis, only the two variables being studied are varied while keeping all other values fixed as given in tables V and VI.

t_write: Please refer to table VI, column 2. The 80386 microprocessor requires a minimum of 2 clock cycles to complete a read or write memory operation, if it is operating in the pipeline mode. [26], i.e., 125 ns which equals 1.25 bus cycles for the level 1 cache. All the memory chips at levels > 1 are assumed to take 4 bus cycles to complete a write.

Send_data(i) and bus_access_time(i)

In case of a miss, the cache fill operation requires a minimum wait of 1 microsecond in addition to the standard two cycle read [26]. We assume that the main memory supplies the 16 byte block in not more than 4 bus cycles ($\text{Send_Data}(2) = 4$) and the rest of the time is spent in accessing and gaining control of the bus, i.e., 6 bus cycles ($\text{bus_access_time}(1) = 6$).

Supply_requesting_cache(i) and Update_parent_cache

Using burst mode, a 16 byte block of data is transferred over the bus in not more than 3 bus cycles.

P_read and P_write. It is assumed that 80% of all accesses are reads and 20% are writes [25].

TABLE VI
INPUT VALUES TO THE MODEL: SET II
(IN BUS CYCLES)

level i	t_write	t_inva- litate	Send Data(i)	block_ replacement(i)	bus_access_ time(i)	supply requesting_ cache(i)	UPDATE_ PARENT_ CACHE
1	1.25	1	1.25	2.80	6.00	3.00	3.00
2	4.00	1	4.00	1.60	6.00	3.00	3.00
3	4.00	1	4.00	1.60	6.00	3.00	3.00
4	4.00	1	4.00	1.60	6.00	3.00	3.00

Effect of Hit_cache[1] and Hit_peer[1]

We will keep all the parameters fixed at values as shown in Tables V and VI and only vary the values of hit_cache[1] and hit_peer[1]. Hit_cache[1] will be some value in the range 0.85 - 1.00 and hit_peer[1] will take a value between 0 - 1.00. The results are shown in figure 15.

It is observed that an increase in the value of hit_cache[1] lowers the average access time (self explanatory), but an increase in the value of hit_peer[1] does not have a significant impact in reducing the average access time. The effect of hit_peer[1] is more pronounced when the level 1 cache's hit ratio is lowest.

This means that at level 1, for reasonable values of hit_cache[1], even if there is no sharing of data among the processors connected to the same bus (hit_peer[1] = 0), it does not adversely affect the performance significantly. On the other hand, the greater the sharing among the processors connected to the same level 1 bus, the better.

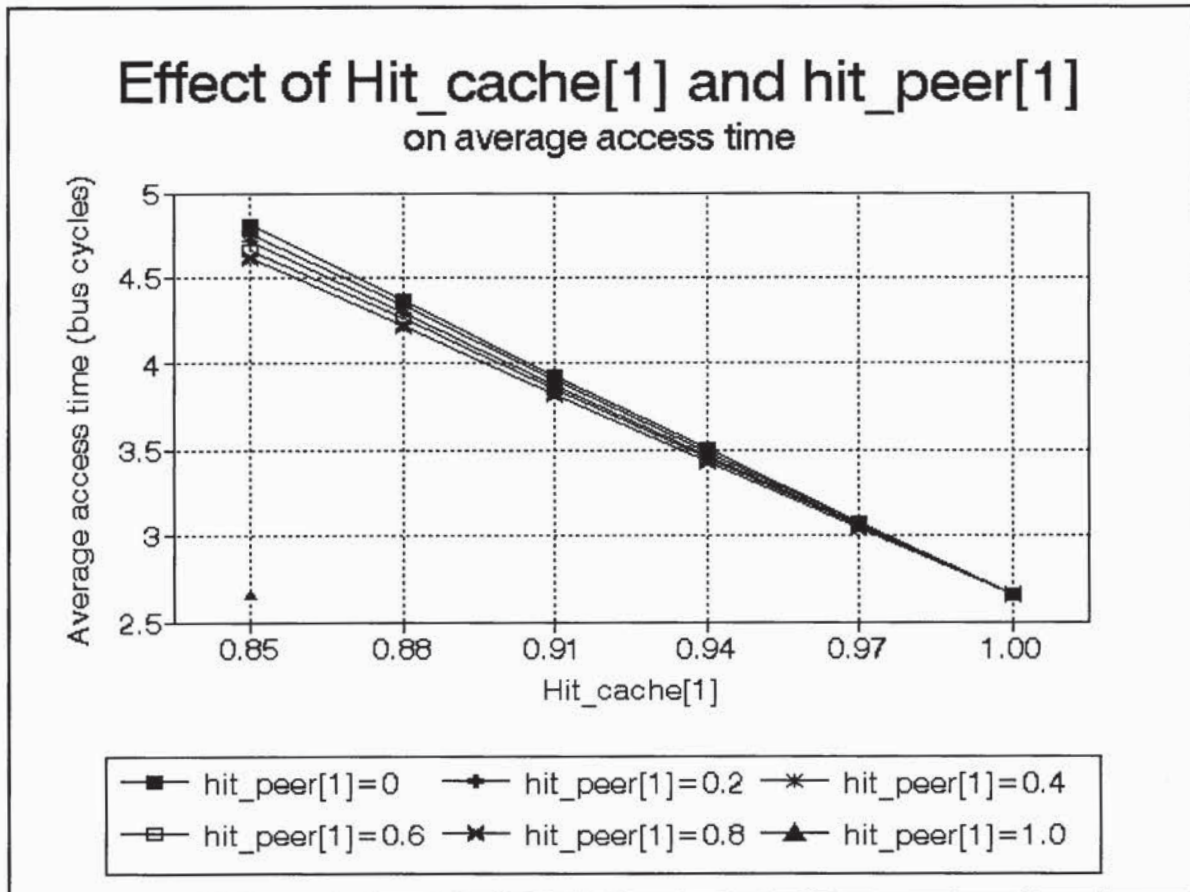


Figure 15. Effect of hit_cache[1] and hit_peer[2] on average access time.

The programmer can then partition the programming task and distribute it in such a way that the sharing is limited to a very small cluster. The least amount of sharing with far away processors will help maximize the computational gains from the multiple processors connected to the same bus. This will help decrease the average access time to fetch a data block.

Effect of Hit_cache[1] and Hit_cache[2]

In this case, the value of hit_cache[1] is varied from 0.80 - 0.99 and hit_cache[2] can have a value between 0.81 and 1.00, while all the other variables take on values as given in Tables V and VI. The analysis is presented in figure 16. The graph highlights the fact that the hit ratio in the level 1 cache seems to be one of the most important factors in reducing the average access time.

A very low value of hit ratio is taken to begin with, i.e., hit_cache[1] = 0.80 which is a very pessimistic estimate. Even at this point, the hit_cache[2] does not impact the average access time significantly .

Increasing the hit ratio at level 1 drastically reduces the time taken to complete a read access (self explanatory) but the case of write access is a bit more complicated. The contribution of the read access to the overall access time decreases and that of the write access increases.

The write access is composed of three different parts; the most expensive of them is the Read-mod operation (write miss, t_write_miss) followed by write hit to clean block (t_write_hit_clean) and lastly write hit to a dirty block (t_write_hit_dirty). As value of hit_cache[1] is increased, the contribution of t_write_miss goes down but that of t_write_hit_clean goes up for a given value of p_clean[1] and p_write. Hence the rate of decline of the write access component is not as steep as the read access. This is shown in figure 17.

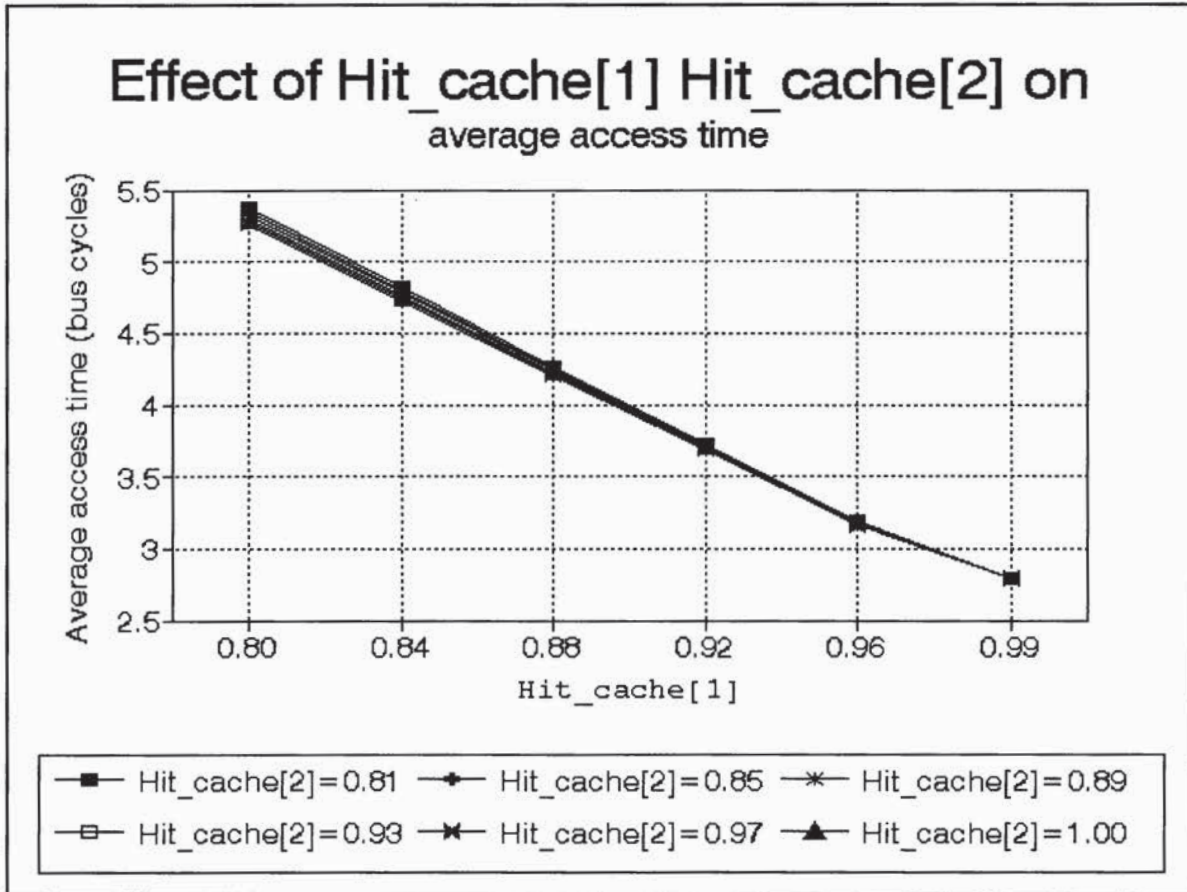


Figure 16. Effect of hit_cache[1] and hit_cache[2] on average access time

Effect of P_clean[1] and P_clean[2] on P_write(1)

P_clean represents the probability that the block is in state clean. The first copy of a block in one of the caches connected to bus (i,j) appears in state clean (chapter 3). Read accesses to the block do not alter its state but a write access changes the state to dirty. Subsequent to the write, if one of the peer caches placed a bus read request for the same block, the cache owning the block in state dirty supplies a copy and then changes the state to clean.

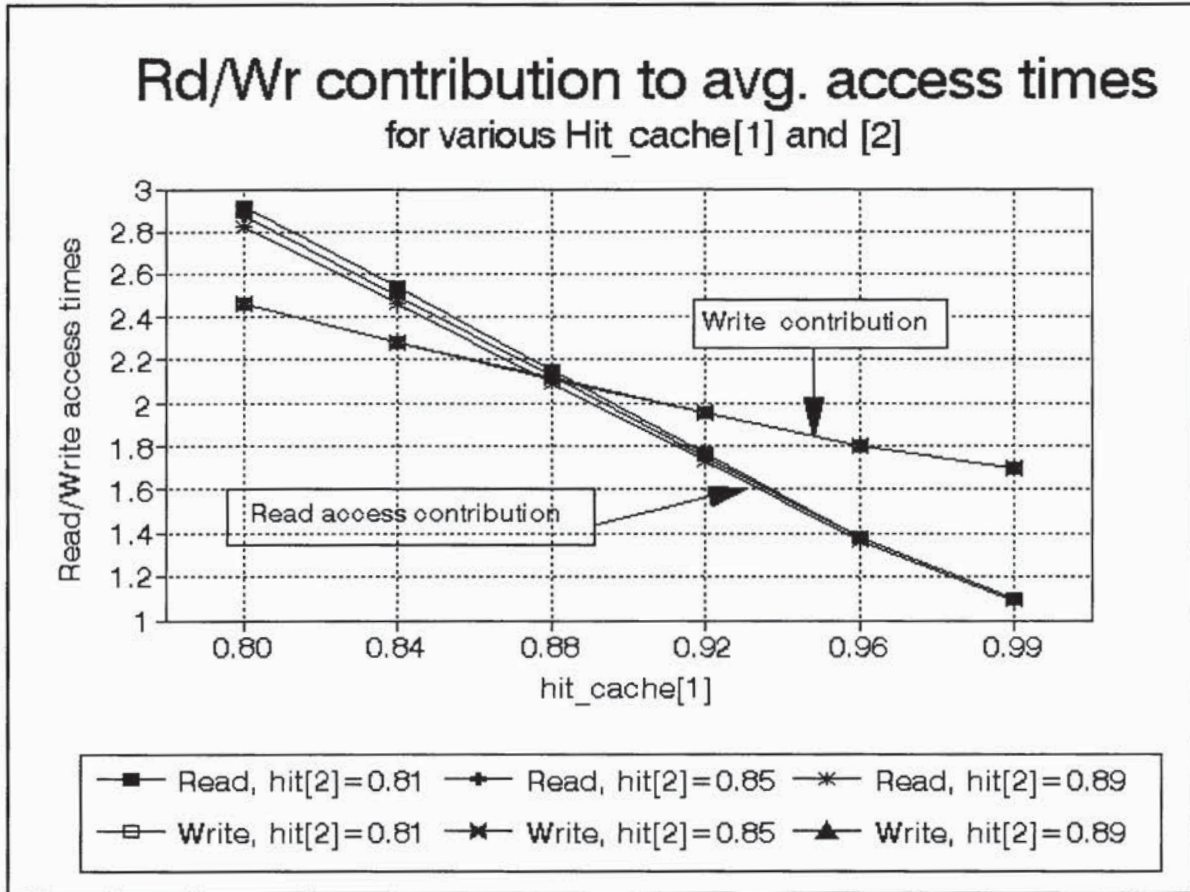


Figure 17. Read/Write access contribution to average access time, for various hit_cache[1] and hit_cache[2].

Thus, a high probability of finding a block in state clean suggests that either there is a relatively high degree of data sharing (read requests from peer caches after a write) or that there were no write accesses to the block.

Please refer to figure 18 for the analysis. The value of p_clean[1] and p_clean[2] are varied from 0 - 1, i.e., from a scenario of no data sharing to maximum data sharing while other variables are held at values as indicated in Tables V and VI.

This is a relatively simple case in that as the probability of a block being in clean state increases, the invalidations have to go higher in the hierarchy. This increases the time to complete a write access. $P_clean[1]$ and $P_clean[2]$ do not have any impact on the read access because a read access is a hit irrespective of the fact whether the block is in state clean or dirty.

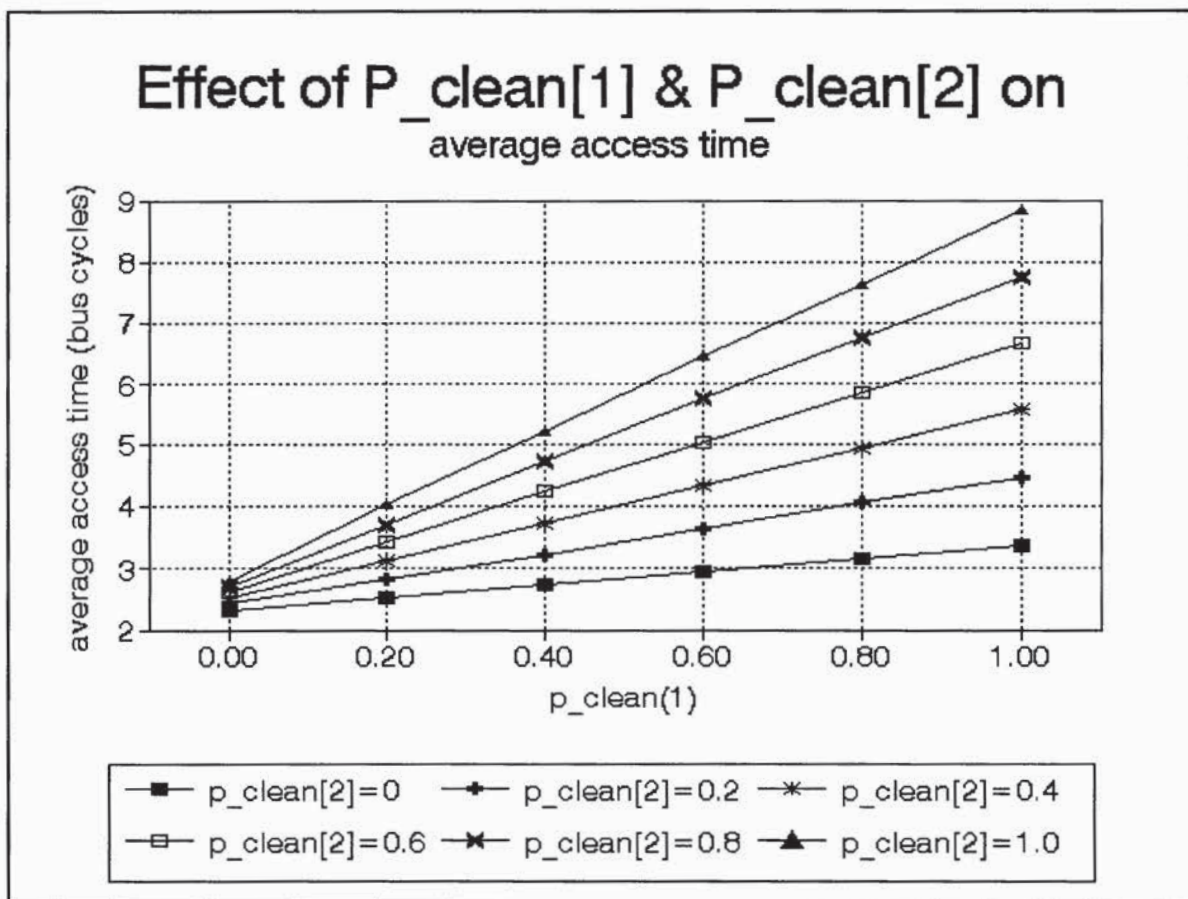


Figure 18. Effect of $p_clean[1]$ and $p_clean[2]$ on average access time.

The increase in the average access time is attributed solely to the write component and in particular to the invalidate process. If there is little sharing of data at different

levels, an invalidate process at level i , where $i < L$, will find the block in state dirty more often and will not have to traverse all the way to the top of the hierarchy. This would directly result in a lower value of average access time.

Effect of Peer_consistent[2] and Peer_consistent[3]

This parameter comes in to play at the time of a B_read, i.e. whenever a read request is placed on the bus by the cache that suffered a read miss. A low value for this parameter means that the peer cache that has to supply the data has a very high probability of finding its copy in state dirty. The up-to-date copy is in one of the descendants and the peer cache must initiate a Recall process to fetch the up-to-date block from its descendant and update its contents. Please refer figure 19 for the effect of these parameters on the recall process.

We have chosen peer_consistent[2] and peer_consistent[3] for our analysis because the higher we are in the hierarchy, the more expensive will be the recall operation. As shown in figure 19, by increasing the value of peer_consistent[2] and peer_consistent[3], we are increasing the probability that the block in the peer cache will find its copy in state consistent. The peer cache would not have to go down to its descendants and fetch the updated copy, thereby reducing the time to perform a recall.

This might give an impression that the TREEBUS architecture will be bogged down with recalls and invalidates and the performance will deteriorate. Please refer to figure 20 for an interesting as well as an encouraging observation.

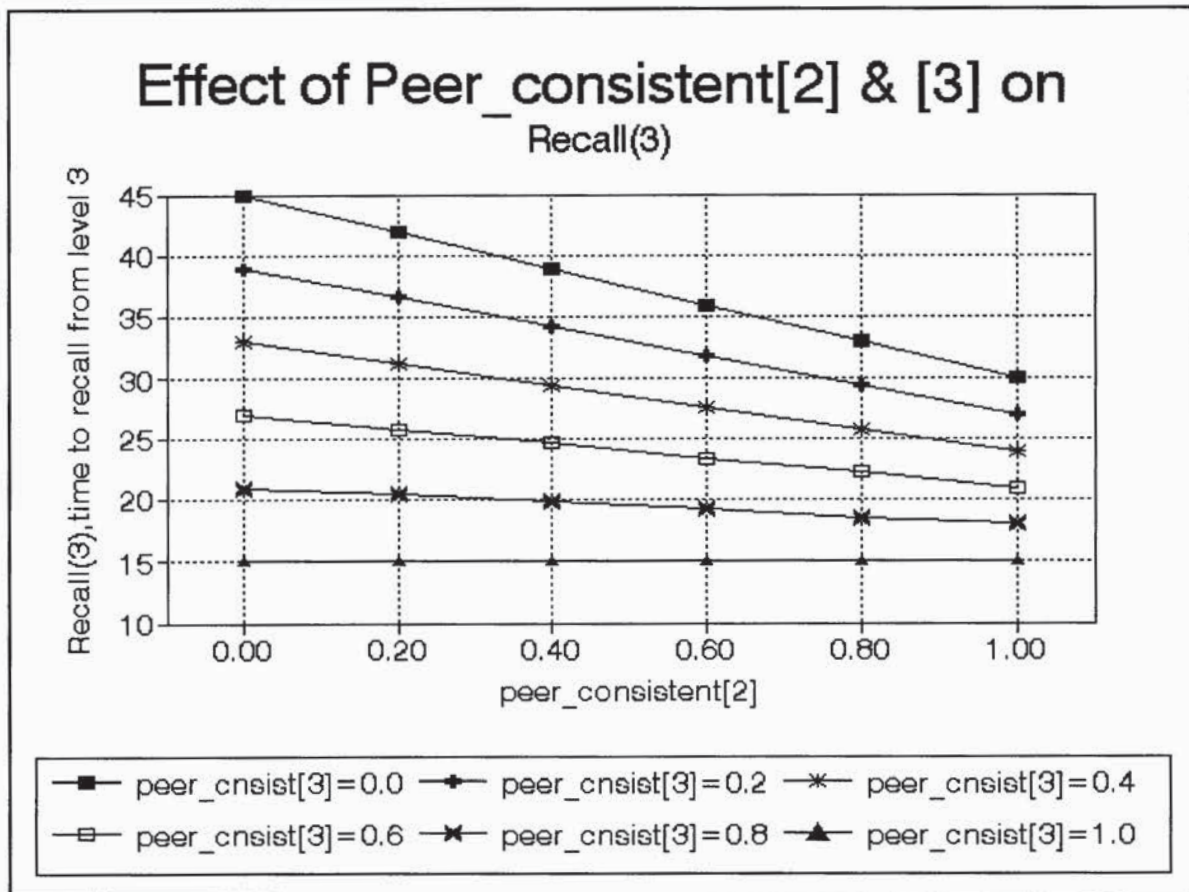


Figure 19. Effect of peer_consistent[2] and peer_consistent[3] on Recall(3).

Even though the worst case scenario for Recall(4) operation in figure 20 requires about 50 bus cycles, i.e., 5 microseconds, the net impact on the bottom line, i.e., average access time, is practically non existent.

The only reason for this odd behavior is that the recall time is weighted by a very small fraction, i.e., $(\text{miss_cache}[1] * \text{miss_cache}[2] * \text{miss_cache}[3] * \text{miss_cache}[4] * \text{hit_peer}[4] * (1 - \text{peer_consistent}[4]))$, which leads to a very small value, practically negligible.

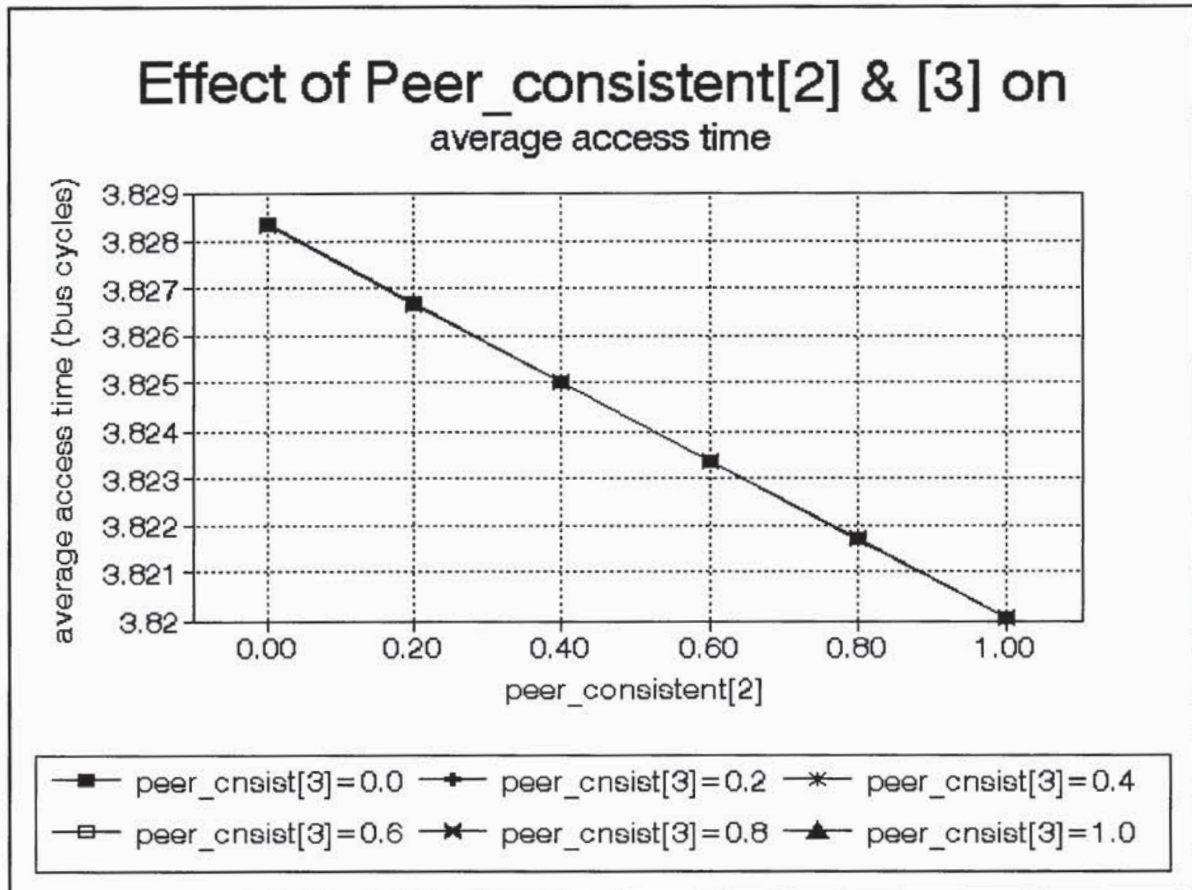


Figure 20. Effect of peer_consistent[2] and peer_consistent[3] on average access time.

This should encourage the designer to go for more levels in the hierarchy, if necessary. Also, if the degree of sharing is low then the number of recall operations will be less thereby further reducing the impact.

Effect of bus_access_time[1] and bus_access_time[2]

It is common knowledge that buses are the bottlenecks in case of a common bus, shared memory architecture systems. The TREEBUS is a bus based architecture and so it is of vital importance to study their impact on system performance.

Please refer to figure 21. In our base case as shown in table V and table VI, we have always taken the bus access time at levels 1 and 2 as 6 cycles. Now, we increase the value of bus access time at level 1 to 36 bus cycles and that at level 2 to 26 bus cycles. The traffic is very high on the level 1 bus, because of the small size of caches at level 1. As we go higher in the hierarchy, the bus traffic should reduce and hence the time to access the bus.

A doubling of the bus access time at level 1 in figure 21 increases the average access time by 33% only. This means that if we have large caches that can arrest the processor's transactions and prevent as many accesses as possible from using the bus, the cache will be able to dwarf the impact of bus accesses on the average access time. This point is proved in Figure 22.

A higher value of hit_cache[1] can arrest the rate of increase of average access time due to the bus accesses. The slope of the graph in figure 22 is less than that in figure 21. Please compare a particular case in figure 21 with bus_access[2]=6 in any graph in figure 20 where the hit_cache[1] > 0.91, and it is clear that a high hit_cache[1] value does help.

Please note that even at hit_cache[1] = 1, the system needs about 8 bus cycles on an average to complete an access. The read access takes only 1.25 bus cycles and the

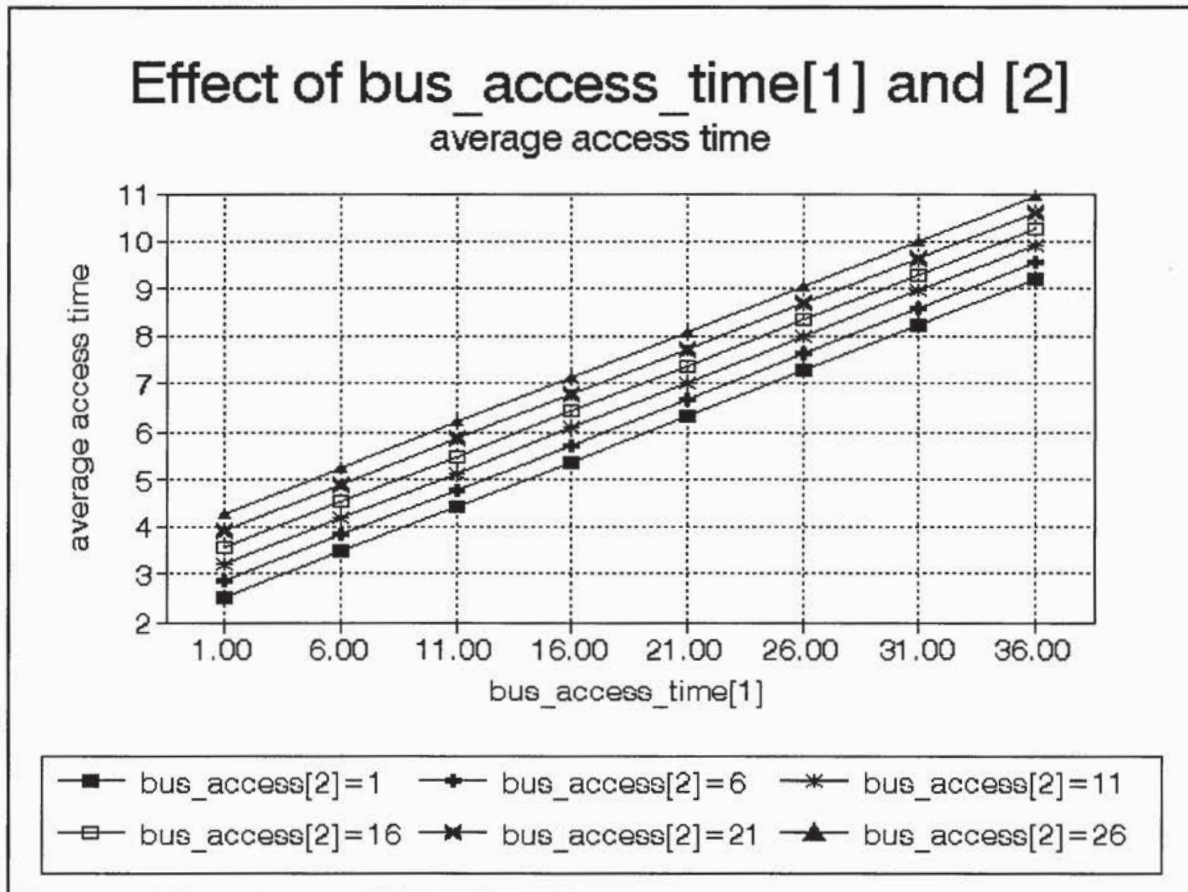


Figure 21. Effect of bus_access_time[1] and bus_access_time[2] on average access time.

rest is the result of write accesses. The only way to reduce the contribution of the write component is by reducing the invalidation overhead. If the invalidation process found the block to be invalidated at higher levels in state dirty, it would not have to go until the top of the hierarchy was reached. This would reduce the number of bus accesses at higher levels and reduce the time to complete a write access. This scenario can be realized by localizing the data sharing pattern to the lower levels of the hierarchy.

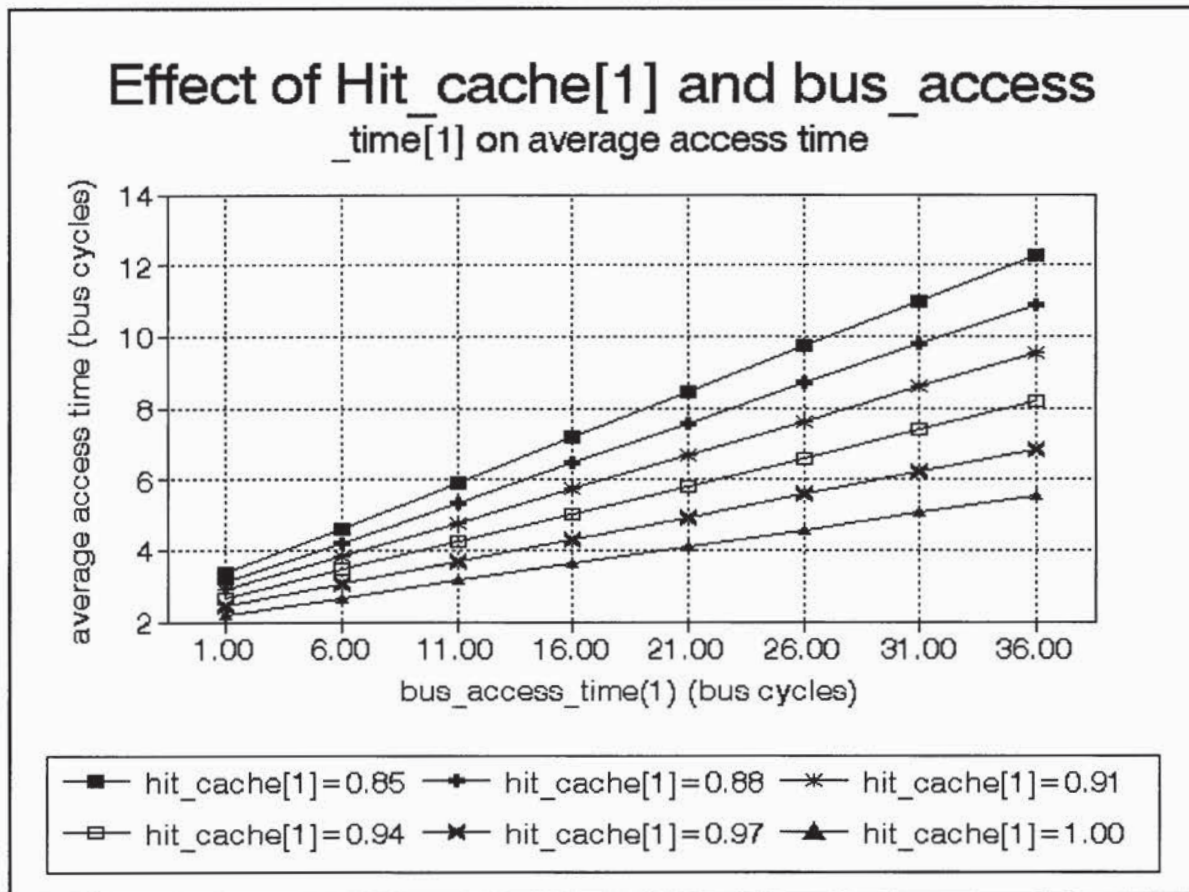


Figure 22. Effect of hit_cache[1] and bus_access_time[1] on average access time.

Example: Consider a three level memory hierarchy with 4 buses at level 1, 2 at level 2 and 1 at level 3 as shown in figure 23. Ideally, we should partition the task into four parts such that all the shared data was localized to each bus, e.g., processors connected to bus (1,1) do not share any data with buses (1,2) through (1,4).

If we achieve the above goal, then except for the first write access to the block, level 3 bus would not see a invalidation for any subsequent writes to the same block.

This is because the copy at level 3 will always be in state dirty after the first write till it is replaced to make room for another incoming block.

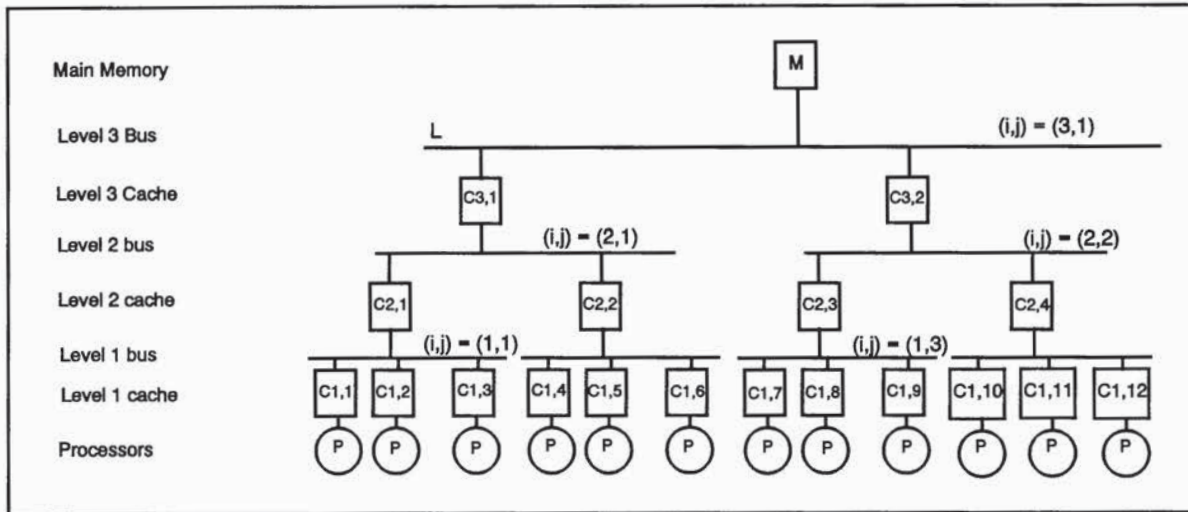


Figure 23. The TREEBUS architecture.

By following a typical course of events, the above paragraph should become absolutely clear.

1. Assume at first, the processor connected to C1,12 suffers a write miss. A copy of the block comes in (C1,12), (C2,4) and in (C3,2) in state clean during the Read part of the Read-Mod operation.

2. During the modification phase of the Read-Mod operation, The invalidate request is sent all the way to the top and the state of the same block in (C1,12), (C2,4) and (C3,2) is now dirty.

3. Next, one of the peer caches of (C1,12) cache, connected to (1,4) bus suffers a read miss. C1,12 supplies a copy and the state of the block in C1,12 and peer cache

is in state clean. Also, the copy in C2,4 gets updated and the state changes to clean. Please note that the block in C3,2 is still in state dirty, and this is the point we were trying to put across.

4. Finally, the processor connected to C1,12 issues a write command to the same block for the second time. The invalidate process has only to go till (2,2) bus because it finds that the block in C3,2 is in state dirty and hence gets completed at the level 2 bus itself.

If this block was shared with the caches connected to say (1,2) bus, then this block in C3,2 cache would have been in state clean, causing the invalidate to go all the way to the top, i.e., (3,1) bus. We have saved some very valuable bus cycles here because of localized data sharing.

Effect of hit_cache[1] on processor's bandwidth requirements

The 80386 processor running at 16 Mhz will need a word of data every 2 cycles (minimum) when running in the pipeline mode. This is a very pessimistic estimate in that the processor completes fetching, decoding and executing the instruction every two cycles. However, this allows us to put more load on the bus and test its response. In reality, the processor would never use the bus so frequently.

In Figure 24, we increase the value of hit_cache[1] and see the effect on the processor's bandwidth demand. Increasing the hit ratio at level 1 reduces the number of times the processor needs to use the bus per unit time. Thus, the processor's net bandwidth demand is reduced.

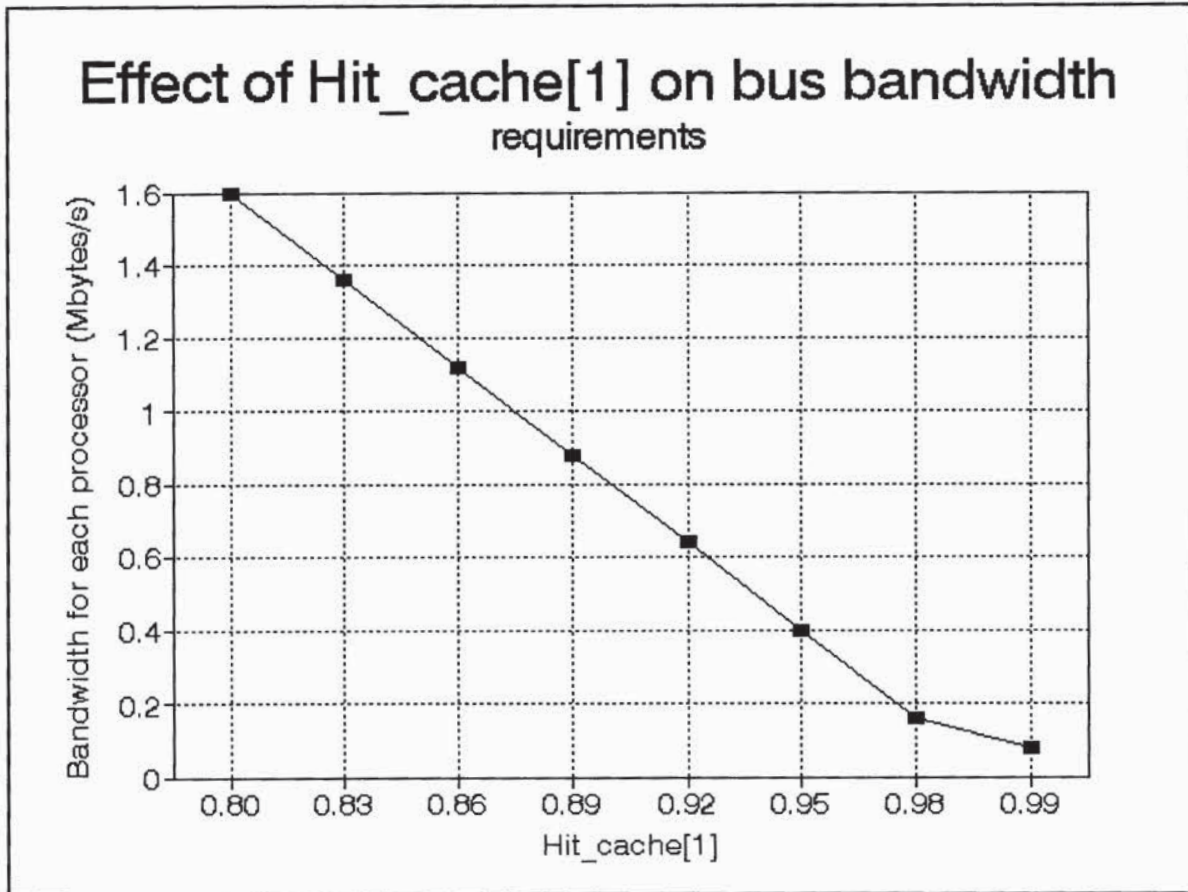


Figure 24. Effect of hit_cache[1] on bus bandwidth requirements.

This suggests that we should have as large a cache as possible before the level 1 bus, so as to increase the effective available bus bandwidth and also reduce the average access time.

EFFECT OF MLI FACTOR (α) ON TOTAL MEMORY SIZE, COST AND COST/PERFORMANCE

The TREEBUS topology being considered for this part of the analysis is as shown in table VII. This translates in to a system with 1024 processors. The authors in [5] have

stated that they have modelled the performance of a system with 2048 processors and are satisfied with the performance. But, even with only 1024 processors the effect of MLI factor is significant on memory size and overall cost. Hence we have used this midway configuration in an attempt to drive home the point that performance is not the only issue that needs to be analyzed. Cost is equally important in our opinion.

At first, we study the effect of α on the total system memory size. This is shown in figure 25.

With an α value of 10, the total size of the memory in the system is approximately 0.8 Terra bytes, out of which the main memory alone occupies 0.610 Terra bytes. As can be clearly seen from the graph, the rate of increase is exponential. Increasing α from 1 to 10 takes the total size to 0.8 Terra bytes, but increasing from 10 to 18 takes the size from 0.8 Terra bytes to approximately 6.8 Terra bytes, an eight fold increase.

The next graph in figure 26 compares the cost ratio of the TREEBUS architecture with that of a system similar to the Sequent Symmetry. If the cost of DRAM is \$ x/byte, the cost of SRAMs is assumed to be \$ 10x/byte. For a particular value of $\alpha = 10$, a TREEBUS system consisting of 1024 processors costs approximately 3000 times more than a Sequent Symmetry with 30 processors. A 34 times increase in the number of processors results in a 3000 times increase in the cost of the memory hierarchy for the TREEBUS system. Also, the rate of cost increase accelerates as the number of processors and number of levels increase.

TABLE VII
TOPOLOGY FOR GRAPHS IN FIGURES 24-25

Number of buses at level 1, N_1	64
Number of buses at level 2, N_2	8
Number of buses at level 3, N_3	2
Number of buses at level 4, N_4	1
Branching factor at level 1, $n_{1,j}$	16
Branching factor at level 2, $n_{2,j}$	8
Branching factor at level 3, $n_{3,j}$	4
Branching factor at level 4, $n_{4,j}$	2

Figure 27 shows a cost/performance ratio comparison between the TREEBUS architecture and a Sequent Symmetry system for different values of α and Hit_cache[1].

The performance is actually the Speedup, defined as:

$$\text{Speedup} = (1/t_{\text{average_access_time}}) \times \text{Number of processors in the system} \quad (24)$$

Ideally, the value of Speedup should be as large as possible and the value of cost/performance ratio should be as small as possible.

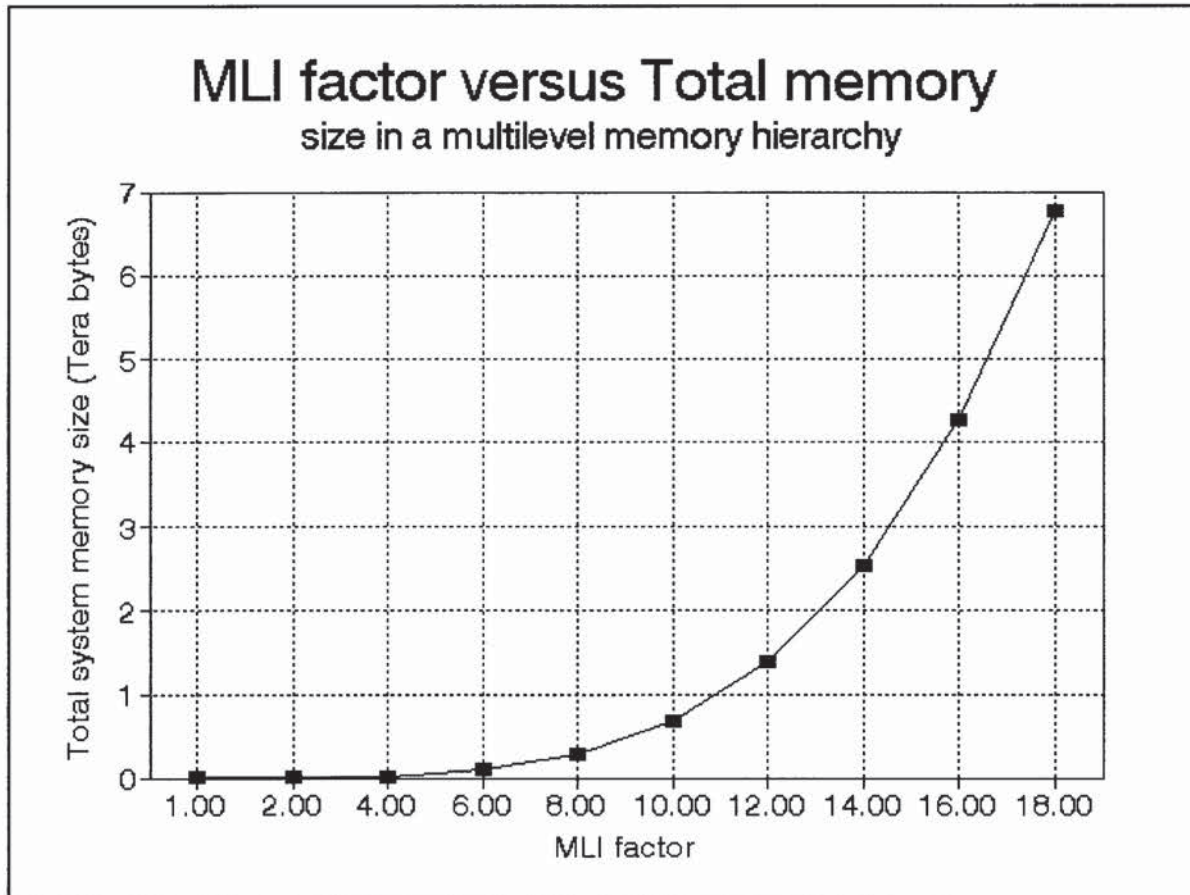


Figure 25. MLI factor versus Total memory size.

A very small fraction would mean that the TREEBUS architecture can deliver higher performance at a lower cost as compared with Symmetry. If the value of both the cost/performance ratios were 1, then the architectures would have been considered equal in terms of cost/performance comparison. A value of half would mean that the TREEBUS architecture can deliver twice the performance as compared to the Sequent Symmetry system.

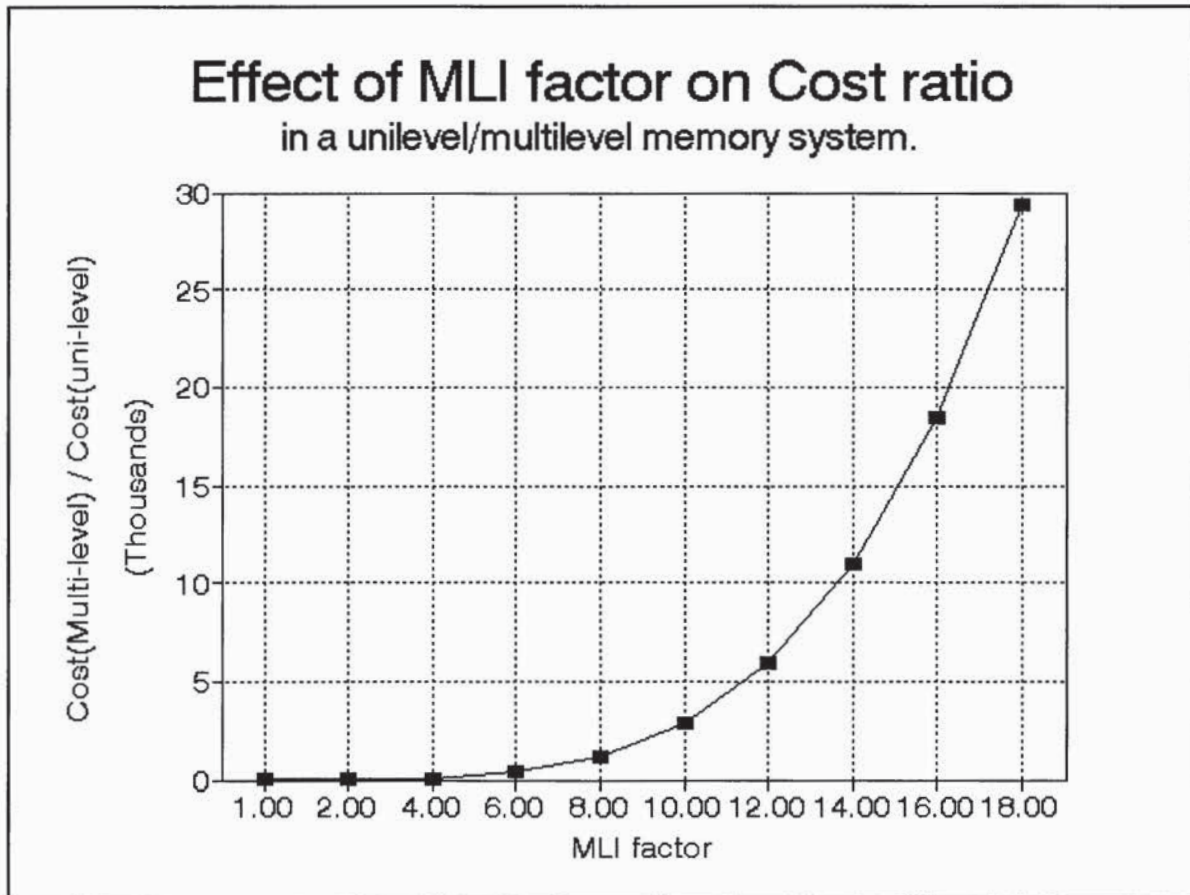


Figure 26. Effect of MLI factor on Cost ratio between unilevel and multilevel memory hierarchy.

The values used for bus access time at different levels in the hierarchy are the lowest possible, i.e., the analysis is based on the most optimistic figures. The performance figures are overstated (lower average access times) and thus the cost/performance ratio is understated.

The value of $\text{Hit_cache}[1]$ is varied from 0.80 to 0.99. We have taken very high values for $\text{hit_cache}[1]$ to demonstrate that α dominates the cost_performance comparison figures. Even for $\text{Hit_cache}[1] = 0.99$, this ratio is around 100 for $\alpha=10$. For $\alpha=18$,

this value shoots to around 900. It seems like that we are not getting a good return on our investment.

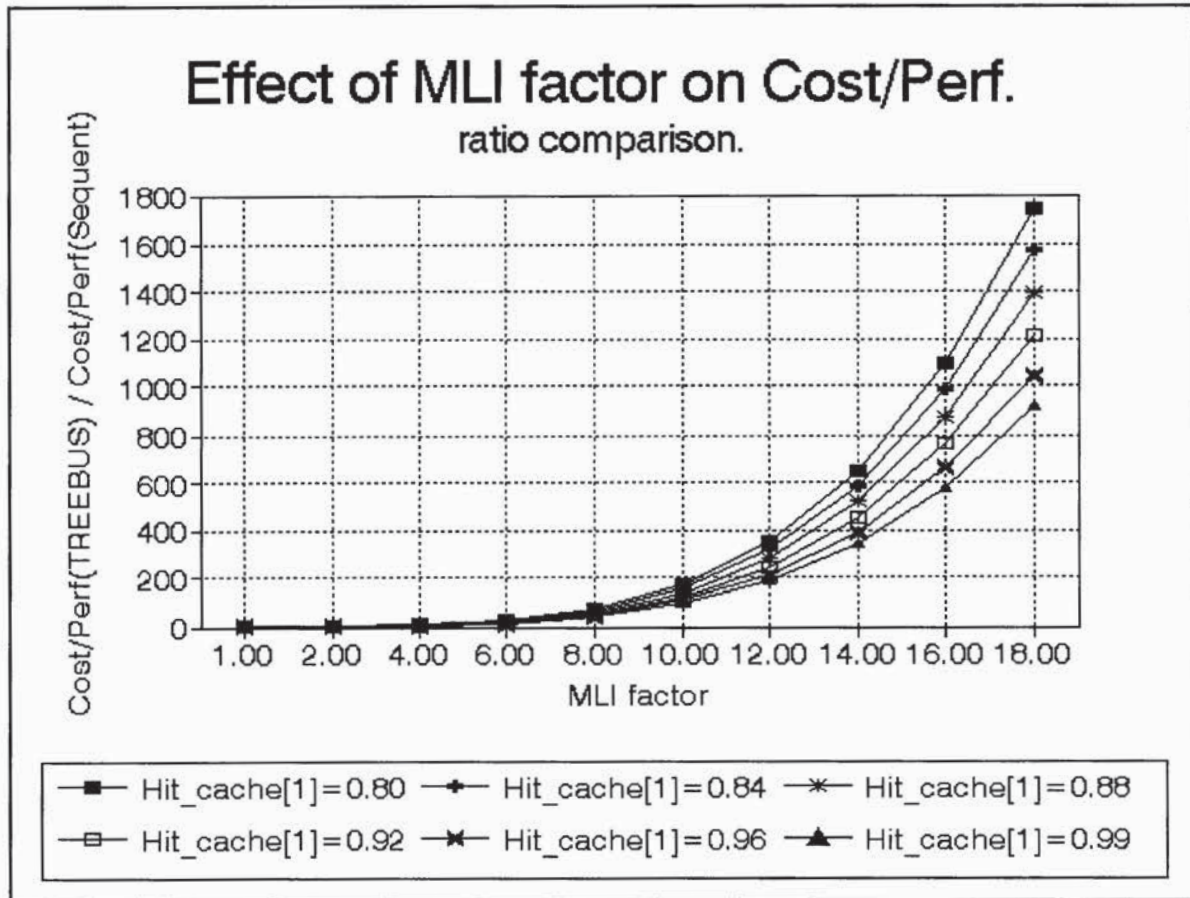


Figure 27. Effect of MLI factor on Cost/Performance ratio comparison between unilevel and multilevel memory hierarchy.

An alternate way of defining performance could be the maximum number of processors that the designer can attach to the level 1 bus without saturating it. This is because, the more processors, the greater the value of Speedup.

The sustained bus bandwidth for the Symmetry's system bus is 53.33 Mbytes/sec. For $\text{hit_cache}[1] = 0.92$, each processor's bus bandwidth demand is 0.64 Mbytes/sec, which translates into a maximum of $(53.33 / 0.64) = 83$ processors that can be connected to the system bus, assuming that there is no I/O traffic on the bus. If the value of $\text{hit_cache}[1]$ is increased to 0.98, the processor's bandwidth demand falls to 0.16 Mbytes/sec, which translates into a maximum of 333 processors that can be connected to the level 1 bus without saturating it.

Figure 28 shows the results using this approach to compare the Sequent Symmetry and TREEBUS architectures. The only time the TREEBUS performs better than the Symmetry is when the value of MLI factor, α , is less than 3. A rule of thumb is that value of α should be at least 10 to have high hit ratios for the caches at all levels. The Symmetry outperforms the TREEBUS in this comparison too.

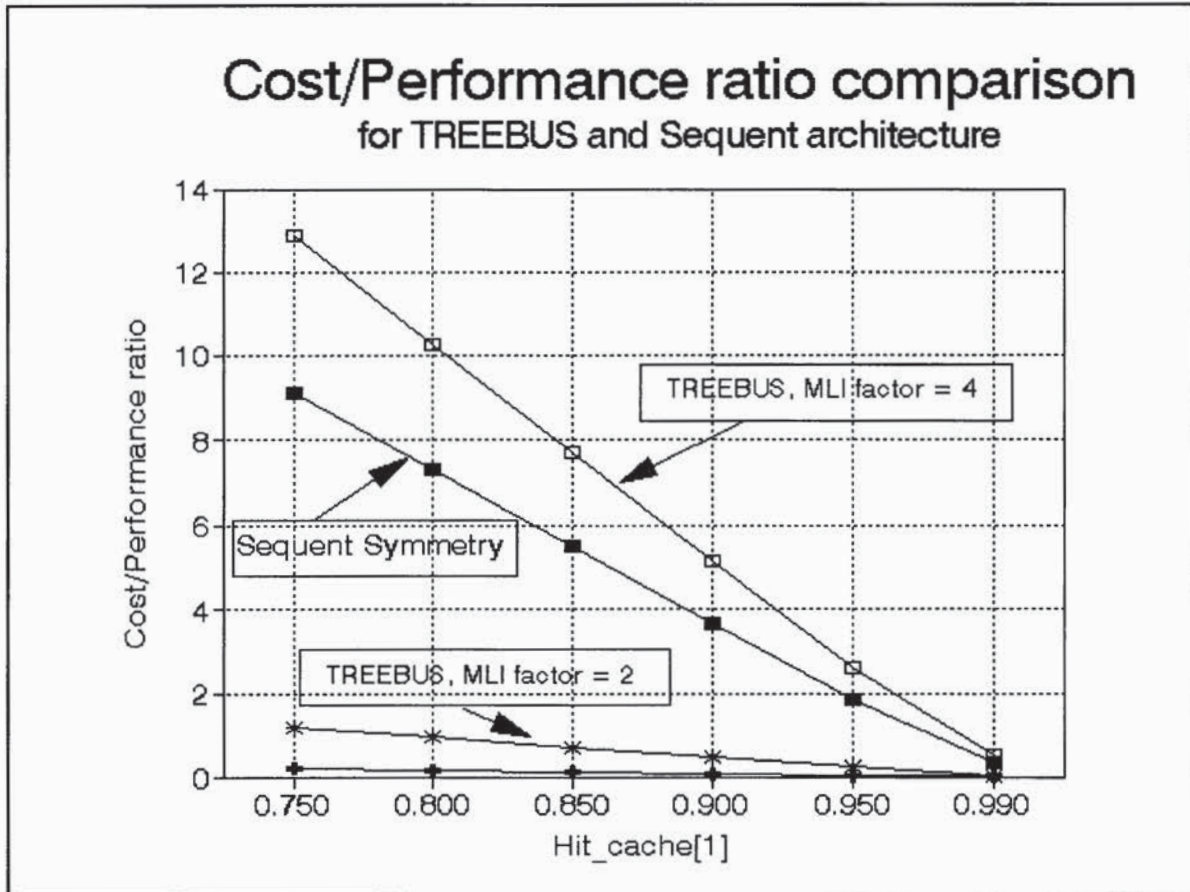


Figure 28. Cost/Performance ratio comparison between TREEBUS and an architecture similar to Sequent's Symmetry.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

In this thesis, we have analyzed the TREEBUS architecture in great detail. At the same time, we have presented a balanced view of the TREEBUS architecture, in that we have not allowed ourselves be carried away by the performance metrics alone. We looked at the cost ratio and cost/performance ratio and it is surprising that the cost aspect has been totally absent in the research conducted so far.

The model that we have developed is reasonably accurate and computationally efficient, allowing us to analyze the architecture in great detail within a very small period of time.

CONCLUSIONS

Maximize hit_cache[1]

The designer of a TREEBUS memory system should primarily focus on maximizing the hit ratio in the private caches. A high value of hit_cache[1] can shield the average access time from the adverse effect of most of the other parameters, e.g., bus access times at higher levels, etc.

The private cache for each microprocessor could be a single level or two-level hierarchy. For today's high performance microprocessors, a two level private cache hierarchy is an absolute must.

This key conclusion is intuitively obvious to every memory system designer but we have reached this decision after detailed and extensive analysis, considering many different parameters that could have adversely affected the system performance.

Figure 16 clearly demonstrates that hit_cache[1] dominates hit_cache[2]. Figure 19 shows how expensive a recall operation can be, but Figure 20 shows that this does not impact the average access time. The effect of bus access time at level 1 and level 2 are shown in figure 21 and Figure 22 shows how increasing hit_cache[1] can mitigate the impact.

Maximize localization of data sharing

This will help us maximize the performance of the TREEBUS architecture. We know that write accesses to a clean block and write misses can wreak havoc on the average access time (shown in Figures 17 and 18). The probability of write misses can be reduced by large caches, but the invalidation process depends solely on the degree of sharing of data, i.e., the invalidation process has to traverse as high as there are clean blocks in the hierarchy.

If a copy exists in state clean, the invalidation request goes all the way to the top and then traverses down, which make it the most expensive operation performed in the system. Localization will ensure that the invalidation request has to go up only a few levels, thereby reducing the average access time.

Cost of the memory hierarchy is a limiting factor

We feel that this is one factor that can easily limit the popularity of the TREEBUS architecture. The size of the main memory virtually explodes beyond manageable proportions for a large size system (1024 - 2048 processors).

As a rule of thumb, MLI factor, α should be equal to 10; using this value for α , the total memory size is around 0.8 Terra bytes. With $\alpha=18$, the size approaches 7 Terra bytes. Ideally, the designer would like α to be as high as possible to maximize the hit ratio at a particular level, but with this architecture the designer's hands are tied.

To decrease the total memory size, α should be lowered. This action would increase the invalidations in the system as a result of increased replacements of blocks from the higher levels to make room for the incoming new blocks. Please note, to enforce MLI, whenever a block is replaced from a cache connected to the level (i,j) bus, all the copies in its descendants need to be invalidated.

TREEBUS still holds promise

All the analysis should not discourage a designer from designing systems around this architecture.

a. It is obvious that TREEBUS cannot compete with distributed memory architecture systems because of the explosion in the size and hence the cost of main memory, but there is a wide performance gap between a single, common bus based, multiprocessor system (e.g., Sequent's Symmetry) and the large scale commercial distributed memory systems, e.g., Intel's Paragon. A niche market definitely exists for the TREEBUS architecture.

b. A very strong point in the TREEBUS's favor is that even for very high bus access times at levels 1 and 2 ($\text{bus_access_time}(1) = 71$ bus cycles and $\text{bus_access_time}(2) = 51$ bus cycles), the average access time is around 20 bus cycles using the values as indicated in tables V and VI.

By using the high performance buses available today, 20 bus cycles translate into an average access time of only 600 ns (using a 33 Mhz bus). The new local buses, e.g., PCI from Intel and VL-bus from VESA [28] can run at rates as high as 66 Mhz. An average access time of around half a micro-second for a multi-level, multi-processor architecture is indeed impressive when compared with average access times of message passing architectures.

c. This is also an attractive architecture from a programmer's point of view because it is easy to parallelize applications written on shared memory machines or transfer applications and programs written for common bus based systems.

VALIDATION

We first tested the model with realistic numbers that were taken from the Sequent Symmetry Technical Summary book. The results matched our expectations totally and only then was the model scaled up for a multi-level multi-processor system. The results from the scaled up version match our intuition and they were also validated by doing manual calculations for a particular set of values. There is no doubt about the

accuracy and correctness of the model, but the way we have generated data for some variables needs further attention in the future. This is discussed in the Future work section.

FUTURE OPTIONS FOR THE DESIGNER

After running the model, the designer has a reasonably fair idea about the characteristics of the system. To get more detailed information, the designer has basically three other options, which are discussed in the following sub-sections.

Develop a model using principles of queuing theory

This would involve mapping the higher level parameters to lower level parameters. In our opinion, the results would not be much more accurate than the ones presented in our model. The researchers in [5] did model using principles of queuing theory and higher level parameters.

Presently, there are no means of measuring these parameters and the designer has to take the best guess approach or sweep the parameters just as we did with lower level parameters.

Simulate the model

The next option would be to simulate the entire system. This is no easy task either, because of the number of processors and the size of the caches and main memory. The designer would need vast computing resources to perform the simulation.

At best, one could simulate a small size system, e.g., a 12 processor configuration as shown in figure 1 to have a better understanding and then extrapolate the results.

Build a prototype

This would be the last option. The researchers at University of Wisconsin, Madison obtained initial results from an analytical model of the Wisconsin Multicube [30] and then started developing a prototype of the system. The reason being that there is very little information available regarding memory reference behavior of parallel programs.

FUTURE WORK

This model lays a solid groundwork for more thorough analysis of the TREEBUS architecture in the future. The model is a good first step and there is scope for further improvement.

Incorporate memory contention

In our model, the memory accesses are assumed to be contention free, i.e., no two write requests are assumed to come to the same cache at the same time. This is never the case in a multi-processor system.

At present, we address this issue by increasing the value of bus access time at various levels and attributing this extra time to contentions while accessing the bus or accessing the same block in the cache.

In case of a conflict, only one processor's request is allowed to proceed, the other processor reissues the request and suffers more delay in accessing the bus.

Model Bus access time accurately

The model in its current state treats Bus_access_time as a sweep variable, but to have a robust model, we should have an accurate equation for deriving its value.

Map higher level input parameters to low level parameters. The higher level parameters would relate closely to the application program characteristics, e.g., frequency of processor reads to a block between writes, frequency of invalidations from processors connected to a different bus at different levels, etc. Using this approach, the results from the model would be easy to interpret.

With our model, the designer would have to do some thinking to interpret the results, i.e., what are the factors that can lead to higher values for p_clean[i]? The answer would be low frequency of writes (because the block always comes in state clean in response to the read request) to the block or high frequency of reads by peer caches between writes.

Make the coherence protocol efficient. The coherence protocol as suggested in [5] can be modified. In the existing protocol, a write hit to a clean block in level i cache generates invalidations on the level i bus, even though this cache could be the only one with a copy. The processor directory as implemented can incorporate an additional state without increasing the implementation overhead and eliminate the unnecessary invalidation cycle.

Include a second level private cache. The proposed TREEBUS architecture [5] has been designed around a single level private cache. A second level private cache is an absolute must for a high performance system to perform at its best. This large secondary cache will not only improve the hit ratio but also reduce the overall access time. This architecture needs larger, faster and smarter memory sub systems that will serve as the level 2 private cache.

Proposed solution: Cache-DRAMs are commercially available today [27] and have a small, extremely fast SRAM in front of a large DRAM. In case of a hit, the SRAM (cache hit reads = 10 ns) can match the CPU cycle time, but in case of a miss, only one normal DRAM access (70 ns - 80 ns) is needed for a cache line fill operation. In the very next cycle, the chip can supply the data at 10 ns. The worst case scenario of two back-to-back cache misses takes 280 ns, because of the DRAM cycle time.

Cache-DRAMs can help in reducing the overall memory cost, reducing the average access time and increasing the available bus bandwidth, thereby allowing the use of more processors and also freeing up bandwidth for input output operations.

SUMMARY

The TREEBUS architecture beyond doubt has the potential to deliver high performance with a reasonable cost tag (cost/performance) for medium sized systems (128 - 256 processors).

The recent developments in the memory technology can be used very effectively in the TREEBUS design to maximize the performance (deliver very high hit ratios) while keeping the overall cost of the memory sub-system to a reasonable level. T h e

model developed is an effective and a reasonably accurate one (provides a great first cut estimate) and can be made more robust with the proposed improvements and modifications. It can also be used as a teaching tool in a case-study context, to show and explain the interaction of multiple parameters on the overall system performance.

REFERENCES CITED

1. Ron Wilson, Senior Editor, "Intel 80486 carries complex instruction set to RISC speeds," *Computer Design*, 11 18-22, May 1, 1989.
2. P. Stenstrom, "A Survey of Cache Coherence Schemes for ~~for~~ Multiprocessors," *Computer*, Vol. 23, No. 6, June 1990, pp. 12-24.
3. H. Cheong and A. Veidenbaum, "A Cache coherence Scheme With Fast Selective Invalidation," *Proc. 15th Int'l Symp. Computer Architecture*, 1988, pp. 299-307.
4. Per Stenstrom, "Reducing contention in Shared_memory Multi-processors," *IEEE Computer*, November 1988, pp 26-37.
5. Mary K. Vernon, Rajeev Jog and Gurindar S. Sohi, "Performance Analysis of Hierarchical Cache Consistent Multiprocessors," *Performance evaluation 9*, 1988/89, pp. 287-302.
6. Jean-Loup Baer and Wen-Hann Wang, "Multilevel cache hierarchies: Organizations, Protocols and Performance," *Journal of parallel and distributed computing* 6, 1989, pp. 451-476.
7. Digital Bus Handbook, Joseph Di Giacomo, McGraw Hill Publishing Company, 1990.
8. The Multibus Design Guidebook, James B. Johnson and Steve Kassel, McGraw Hill Publishing Company, 1984.
9. Ron Wilson, "Intel 80486 carries complex instruction set to RISC speeds," *Computer Design*, pp 18-22, May 1, 1989.
10. Warren Andrews, "Static RAMs race to keep up with RISC," *Computer Design*, pp 59-66, April 1, 1989.
11. Ron Wilson, "68040 moves toward RISC camp with redesigned pipelines caches," *Computer Design*, pp 22, May 1, 1989.

12. Jim Handy, "Practical design techniques for today's RISC and CISC CPU's," *Electro International Conference Record*, pp 283-289, April 16-18, 1991.
13. A. W. Wilson, "Hierarchical Cache/Bus architecture for Shared Memory Multiprocessors," in *Proc. 14th Annual Symposium on Computer Architecture*, Pittsburgh, PA, pp. 244-252, June 1987.
14. BiCMOS and CMOS Data book, Cypress Semiconductor, pp. 9.152-9.157, March 1992
15. Rajeev Jog, G. S. Sohi, and M. K. Vernon, "The Treebus architecture and its analysis," *Computer Sciences Tech. Rep. #747*, University of Wisconsin-Madison, Madison, WI 53706, 1988.
16. Ron Wilson, "Cache controllers tread a rocky path toward integration," *Computer Design*, pp 99-108, November 1, 1990.
17. David Chaiken Craig Fields, Kiyoshi Kurihara and Anant Agarwal, "Directory-Based Cache Coherence in Large-Scale Multiprocessors," *IEEE Computer*, pp. 49-58, June 1990.
18. Hoichi Cheong and Alexander V. Veidenbaum, "Compiler-Directed Cache Management in Multiprocessors," *IEEE Computer*, pp. 39-47, June 1990.
19. D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proc. 17th Int'l Symp. Computer Architecture*, pp. 148-159, May 1990.
20. Paragon™ XP/S Product Overview, Intel Corporation, 1991
21. George Watson, "The main event," *IEEE Spectrum*, pp 30, January 1991.
22. Warren Andrews, "Will performance win over sophistication in workstation buses?" *Computer Design*, pp. 78-89, February 1, 1991.
23. Betty Prince, Roger Norwood, Joe Hartigan and Wilbur C. Vogley, "Synchronous dynamic DRAM," *IEEE Spectrum*, pp. 44-47, October 1992.
24. Charles A. Hart, "Dynamic RAM as secondary cache," *IEEE Spectrum*, pp 48, October 1992.
25. Jean-Loup Baer and Wen-Hann Wang, "Architectural choices for Multi-Level Cache Hierarchies," *Technical Report 87-01-04*, University of Washington, Seattle, January 14, 1987.

26. Symmetry Technical Summary, Sequent Computer Systems, Inc., 1987.
27. Richard Quinnell, "4-Mbit DRAM integrates SRAM cache for 10-nsec cache-hit access," *EDN*, pp 77, March 16, 1992.
28. Bob Francs, "Should IS ride the local bus?," *Datamation*, pp 47-49, March 15, 1993.
29. C. P. Thacker, L. C. Stewart and E. H. Satterthwaite, Jr., "Firefly: a Multiprocessor Workstation," *IEEE Transactions on Computers*, 37, 8 (August 1988), 909-920.
30. James R. Goodman and Philip J. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor," in *Proc. 15th Annual Symposium on Computer Architecture*, 1988, pp 422-431.
31. Eugene Levin, "Grand Challenges to Computational Science," *Communications of the ACM*, December 1989, Volume 32, Number 12, pp 1456-1457.

APPENDIX A

READ.C

```

#include <stdio.h>
#include "header.h"
#include "data.h"

float P_read(j)
int j;
{
    float x, y, z = 0;

    /*    If we are at the top of memory hierarchy, then
        main memory supplies data and the function
        send_data() is called.          */

    if( j == LIMIT)
        z = send_data(j);
    else {

        /*    Normal flow of operations, since not at the
            top of memory hierarchy yet.
            HIT TIMINGS          */

        y = send_data(j) ;
        z = ( hit_cache[j-1] * y ) ;

        /*    if we have a miss, the request appears on the
            level 'j' bus in the memory hierarchy.
            MISS TIMINGS      */

        if( hit_cache[j-1] != 1 ){

            /*    This if logic is to ensure that when the request is
                coming down with the block of data, we have taken care
                of the additional bus delay involved. */

                if ( j > 1)
                    x = B_Read(j) + send_data(j) +
                        bus_access_time(j-1);
                else
                    x = B_Read(j) + send_data(j) ;

            /* Mean access time at level j */

            z += ( (1 - hit_cache[j-1]) * x ) ;

```

```

        }
    }
    return z ;
}

```

```

float send_data(k)
int k;
{
    if( k > 1 )
        return 4.00;
    else
        return 1.25 ;
}

```

```

float Block_replacement(l)
int l;
{
    return ( (1 - p_clean[l-1]) * T_write[l] );
}

```

```

float B_Read(n)
int n;
{
    float x = 0;

    if( hit_peer[n-1] != 0 ){
        x = get_from_peer(n) ;
        x *= hit_peer[n-1];
    }

    if( hit_peer[n-1] != 1 ){

```

```

        x += ( P_read(n+1) ) * ( 1 - hit_peer[n-1] ) ;
    }

    x += ( bus_access_time(n) + Block_replacement(n) );

    return x;
}

float get_from_peer(o)
int o;
{
    float x, z;
    float y = 0;

    x = supply_requesting_cache(o) ;

    if(peer_consistent[o-1] != 1)
        y = recall(o); /* If inconsistent, we need to
                        perform a recall */

    z = ( peer_consistent[o-1] * x ) + (
        ( 1 - peer_consistent[o-1]) * ( y + x ) );

    return z ;
}

float recall(tex)
int tex;
{
    float val = 0;

    /* This is done because recall(2) must find a consistent
       copy of the block at level 1. Level 1 is the closest
       to the processor and the data at this level is most
       updated or in state "consistent." Recall(1) makes no sense,
       because such a thing will never happen. We are fine from
       the highest level to level 2 for recall purposes as
       explained above. */

```

```

    if (tex > 1) {

/*      There is an extra term for bus_access_time() in the
        formula below This is because we access the bus once while
        going down to fetch the updated block and when we are going
        up to update the parent cache, we again need to access the
        bus. We cannot hold on to the bus till the complete of
        an operation as this would slow operations down considerably. */

/*      The first assignment to val is for the case when the peer
        cache has a consistent copy */

        val = ( UPDATE_PARENT_CACHE + bus_access_time(tex - 1));

/*      The second part deals with a peer cache with an inconsistent copy */

        val += ( 1 - peer_consistent[tex-2] ) * recall(tex-1) ;

/*      The last part adds the one extra bus access that we have to
        take care of */

        val += bus_access_time(tex);

        return val;

    }

    else

        return (val = 0);

}

float supply_requesting_cache(q)
int q;
{
    return 3.00;
}

float bus_access_time(s)
int s;

```

```
{  
    return 6.00 ;  
}
```

APPENDIX B

WRITE.C

```

#include <stdio.h>
#include "header.h"
#include "data.h"

main()
{
    int bus_level = 1;
    float t_read_access, t_write_access;
    float t_average_access_time;

    t_read_access = P_read(bus_level) * p_read;
    t_write_access = P_write(bus_level) * (1 - p_read);
    t_average_access_time = t_read_access + t_write_access;

    printf("t_read_access_time: %5.4f. \n\n", t_read_access);
    printf("t_write_access_time: %5.4f. \n\n", t_write_access);
    printf("t_average_access_time: %5.4f.\n\n\n",
           t_average_access_time );
}

float P_write(j)
int j;
{
    float temp, t_write_hit_clean, t_write_hit_dirty;
    float temp1, t_write_miss;

    temp = Invalidate(j);
    temp1 = P_read(j);

    t_write_hit_clean = ( hit_cache[j-1] * p_clean[j-1] ) *
                        ( temp + t_write[j-1] );

    t_write_hit_dirty = ( hit_cache[j-1] * (1 - p_clean[j-1]) *
                        t_write[j-1]);

    t_write_miss = (1 - hit_cache[j-1]) * ( temp1 + temp +
                        t_write[j-1] );

    printf("P_read(%d): %f\n", j, temp1);
}

```

```

printf("Invalidate(1) = %f\n",temp);
printf("twm = %f\n", t_write_miss);
printf("t_whc= %f\n", t_write_hit_clean);
printf("twhd = %f\n", t_write_hit_dirty);

temp = t_write_hit_clean + t_write_hit_dirty +
        t_write_miss;
return temp;
}

float Invalidate(k)
int k;
{
    float temp = 0;

    /* If we are at the top of the hierarchy, we do not
       need to go up any further. That's what this "if"
       statement ensures. */

    if ( (k >= 1) && (k < L) )
        temp += ( p_clean[k] * ( bus_access_time(k) +
                                Invalidate(k+1)) ) ;
    else
        temp = 0;

    temp += Invalid_ack(k) + bus_access_time(k) + t_invalidate;
    return temp;
}

float Invalid_ack(l)
int l;
{
    if (l > 1)
        return 1.00;

    /* The time needed is to just send a signal to the lower
       level cache, hence 1 bus cycle is enough */

    else
        return 0;
}

```

APPENDIX C

HEADER FILES

Data.h

```
#define LIMIT 4 /* LIMIT = L + 1; L = 3, a 3 level hierarchy */
#define UPDATE_PARENT_CACHE 3
#define t_invalidate 1
#define p_read 0.80
#define L (LIMIT - 1)
float hit_cache[LIMIT] = { 0.91, 0.95, 0.99, 1.0};
float hit_peer[LIMIT] = { 0.80, 0.75, 0.75, 0.75};
float peer_consistent[LIMIT] = { 1.00, 0.85, 0.85, 0.85};
float p_clean[LIMIT] = {0.30, 0.60, 0.60, 0.60};
float t_write[LIMIT] = {1.25, 4, 4, 4};
```

Header.h

```
float send_data(), B_Read();
float bus_access_time() ;
float block_replacement(), get_from_peer();
float P_read();
float recall();
float bus_access_time();
float supply_requesting_cache();
float update_parent_cache();
float P_write();
float Invalidate();
float Invalid_ack();
```