BUS AND CACHE MEMORY ORGANIZATIONS FOR MULTIPROCESSORS

By

Donald C. Winsor

CSE-TR-19-89

THE UNIVERSITY OF MICHIGAN

COMPUTER SCIENCE AND ENGINEERING DIVISION
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
Room 3402, EECS Building
Ann Arbor, Michigan 48109-2122
USA

ABSTRACT

BUS AND CACHE MEMORY ORGANIZATIONS FOR MULTIPROCESSORS

by
Donald Charles Winsor

Chairman: Trevor Mudge

The single shared bus multiprocessor has been the most commercially successful multiprocessor system design up to this time, largely because it permits the implementation of efficient hardware mechanisms to enforce cache consistency. Electrical loading problems and restricted bandwidth of the shared bus have been the most limiting factors in these systems.

This dissertation presents designs for logical buses constructed from a hierarchy of physical buses that will allow snooping cache protocols to be used without the electrical loading problems that result from attaching all processors to a single bus. A new bus bandwidth model is developed that considers the effects of electrical loading of the bus as a function of the number of processors, allowing optimal bus configurations to be determined. Trace driven simulations show that the performance estimates obtained from this bus model agree closely with the performance that can be expected when running a realistic multiprogramming workload in which each processor runs an independent task. The model is also used with a parallel program workload to investigate its accuracy when the processors do not operate independently. This is found to produce large errors in the mean service time estimate, but still gives reasonably accurate estimates for the bus utilization.

A new system organization consisting essentially of a crossbar network with a cache memory at each crosspoint is proposed to allow systems with more than one memory bus to be constructed. A two-level cache organization is appropriate for this architecture. A small cache may be placed close to each processor, preferably on the CPU chip, to minimize the effective memory access time. A larger cache built from slower, less expensive memory is then placed at each crosspoint to minimize the bus traffic.

By using a combination of the hierarchical bus implementations and the crosspoint cache architecture, it should be feasible to construct shared memory multiprocessor systems with several hundred processors.

To my family and friends

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Advances in VLSI (very large scale integration) technology have made it possible to produce high performance single-chip 32-bit processors. Many attempts have been made to build very high performance multiprocessor systems using these microprocessors because of their excellent cost/performance ratio.

Multiprocessor computers can be divided into two general categories:

- shared memory systems (also known as tightly coupled systems)
- distributed memory systems (also known as loosely coupled systems)

Shared memory systems are generally easier to program than distributed memory systems because communication between processors may be handled through the shared memory and explicit message passing is not needed. On the other hand, shared memory systems tend to be more expensive than distributed memory systems for a given level of peak performance, since they generally require a more complex and costly interconnection network.

This thesis will examine and present new solutions to two principal problems involved in the design and construction of a bus oriented shared memory multiprocessor system. The problems considered in this thesis are the limitations on the maximum number of processors that are imposed by capacitive loading of the bus and limited bus bandwidth.

1.1 Single bus systems

In the most widely used shared memory multiprocessor architecture, a single shared bus connects all of the processors, the main memory, and the input/output devices. The name *multi* has been proposed for this architecture. This architecture is summarized as follows in [Bell85]:

> Multis are a new class of computers based on multiple microprocessors. The small size, low cost, and high performance of microprocessors allow the design and construction of computer structures that offer significant advantages in manufacture, price-performance ratio, and reliability over traditional computer families.

Figure 1.1 illustrates this architecture. Representative examples of this architecture include the Encore Multimax [Encor85] and the Sequent Balance and Sequent Symmetry series [Seque86]. The popularity of this architecture is probably due to the fact that it is an evolutionary step from the familiar uniprocessor, and yet it can offer a performance increase for typical multiprogramming workloads that grows linearly with the number of processors, at least for the first dozen or so.

The architecture of Figure 1.1 can also be used in a multitasking environment where single jobs can take control of all the processors and execute in parallel. This is a mode of operation which is infrequently used at present, so the discussion in this thesis emphasizes a multiprogramming environment in which computational jobs form a single queue for the next available processor. One example of a single job running on all processors in parallel is considered, however, to demonstrate that the same design principles are applicable to both situations.

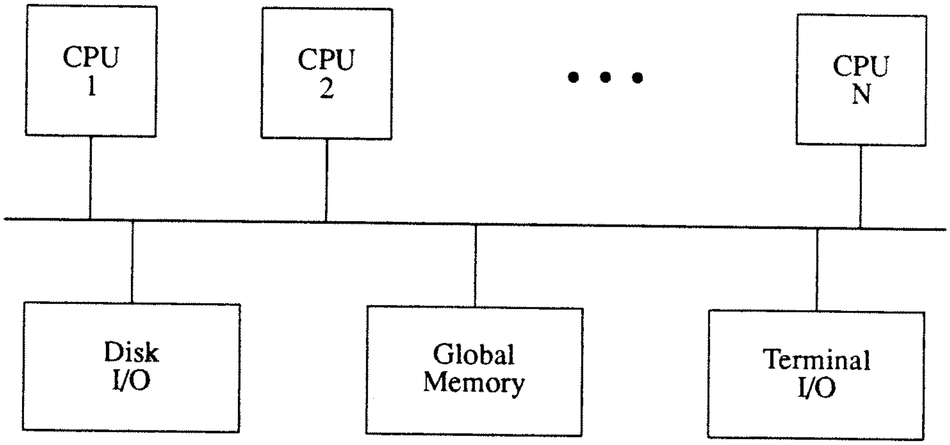


Figure 1.1: Single bus shared memory multiprocessor

1.2 Cache memories

The maximum performance of shared memory systems is extremely sensitive to both bus bandwidth and memory access time. Since cache memories significantly reduce both bus traffic and average access time, they are an essential component of this class of multiprocessor. Figure 1.2 shows a multiprocessor system with a private cache memory for each processor. When using a private cache memory for each processor, it is necessary to ensure that all valid copies of a given cache line are the same. (A cache line is the unit of data transfer between cache and main memory.) This requirement is called the *multicache consistency* or *cache coherence* problem.

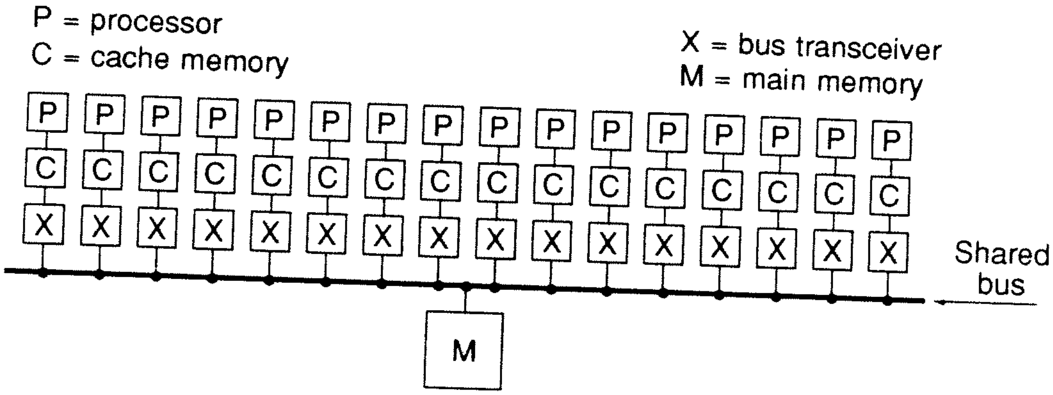


Figure 1.2: Shared memory multiprocessor with caches

Many solutions to the cache consistency problem have been proposed, including software and hardware consistency algorithms, and combinations of both. Software solutions require the compiler or operating system to identify all shared regions of memory and to issue appropriate commands to the caches to ensure consistency. This is typically done by using lock variables to enforce mutually exclusive access to shared regions and having each processor flush its cache prior to releasing the mutual exclusion lock. The large amount of memory traffic generated by the frequent cache flushes makes software solutions impractical for large numbers of processors. Furthermore, the requirement that shared regions be identified by software is difficult to achieve and can add significantly to the complexity of the compiler and operating system.

The most promising hardware solutions to the cache consistency problem require that all processors share a common main memory bus (or the logical equivalent). Each cache monitors all bus activity to identify references to its lines by other caches in the system. This monitoring is called *snooping* on the bus or *bus watching*. The advantage of snooping caches is that consistency is managed by the hardware in a decentralized fashion, avoiding the bottleneck of a central directory. Practical snooping cache designs will be discussed in detail in Chapter 2 of this dissertation.

1.3 Bus electrical limitations

Until recently, the high cost of cache memories limited them to relatively small sizes. For example, the Sequent Balance multiprocessor system uses an 8 K-byte cache for each processor [Seque86]. These small caches have high miss ratios, so a significant fraction of memory requests require service from the bus. The resulting high bus traffic limits these systems to a small number of processors. Advances in memory technology have substantially increased the maximum practical cache memory size. For example, the Berkeley SPUR multiprocessor workstation uses a 128 K-byte cache for each processor [HELT*86], and caches as large as 1024 K-bytes are being considered for the Encore Ultramax described in [Wilso87]. By using large caches, it is possible to reduce the bus traffic produced by each processor, thus allowing systems with greater numbers of processors to be built.

Unfortunately, capacitive loading on the bus increases as the number of processors is increased. This effect increases the minimum time required for a bus operation, thus reducing the maximum bus bandwidth. As the number of processors is increased, a point is eventually reached where the decrease in bus bandwidth resulting from the added bus load of another processor is larger than the performance gain obtained from the additional processor. Beyond this point, total system performance actually decreases as the number of processors is increased.

With sufficiently large cache memories, capacitive loading, driver current limitations, and transmission line propagation delays become the dominant factors limiting the maximum number of processors. Interconnection networks that are not bus oriented, such as multistage networks, are not subject to the bus loading problem of a single bus. The bus oriented cache consistency protocols will not work with these networks, however, since they lack an efficient broadcast mechanism by which a processor can inform all other processors each time it references main memory. To build very large systems that can benefit from the advantages of the bus oriented cache consistency protocols, it is necessary to construct an interconnection network that preserves the logical structure of a single bus while avoiding the electrical implementation problems associated with physically attaching all of the processors directly to a single bus.

General background information on buses is presented in Chapter 2. In Chapter 3, several interconnection networks suitable for implementing such a logical bus are presented. A new model of bus bandwidth is developed that considers the effects of electrical loading on the bus. It is used to develop a practical method for estimating the maximum performance of a multiprocessor system, using a given bus technology, and to evaluate the logical bus networks presented. In addition, a method is given for selecting the optimal network given the electrical parameters of the implementation used.

1.4 Trace driven simulation

To validate the performance model developed in the Chapter 3, simulations based on address traces were used. Chapter 4 presents the simulation models used, the workloads for which traces were obtained, and the results of these simulations.

1.5 Crosspoint cache architecture

In Chapter 5, a new architecture is proposed that may be used to extend bus oriented hardware cache consistency mechanisms to systems with higher bandwidths than can be obtained from a single bus. This architecture consists of a crossbar interconnection network with a cache memory at each crosspoint. It is shown that this architecture may be readily implemented using current VLSI technology. It is also shown that this architecture is easily adapted to accommodate a two-level cache configuration.

1.6 Techniques for constructing large systems

In Chapter 6, a demonstration is given of how hierarchical bus techniques described in Chapter 3 may be applied to the crosspoint cache architecture presented in Chapter 5. The combination of these two approaches permits a substantial increase in maximum feasible size of shared memory multiprocessor systems.

1.7 Goal and scope of this dissertation

As discussed in the previous sections, the bus bandwidth limitation is perhaps the most important factor limiting the maximum performance of bus based shared memory multiprocessors. Capacitive loading of the bus that increases with the number of processors compounds this bandwidth problem. The goal of this dissertation is to provide practical methods for analyzing and overcoming the bus bandwidth limitation in these systems. Current commercial shared memory multiprocessor systems are limited to a maximum of 30 processors. The techniques developed in this dissertation should permit the construction of practical systems with at least 100 processors.

1.8 Major contributions

The following are the major contributions of this thesis:

- A new bus bandwidth model is developed in Chapter 3. Unlike previous models, this model considers the effects of electrical loading of the bus as a function of the number of processors. The new model is used to obtain performance estimates and to determine optimal bus configurations for several alternative bus organizations.

- The results of a trace driven simulation study used to validate the bus bandwidth model are presented in Chapter 4. Performance estimates obtained from the bus bandwidth model are shown to be in close agreement with the simulation results.

- A proposal for a new architecture, the *crosspoint cache architecture*, is presented in Chapter 5. This architecture may be used to construct shared memory multiprocessor systems that are larger than the maximum practical size of a single bus system, while retaining the advantages of bus oriented hardware cache consistency mechanisms.

- A demonstration of how hierarchical bus techniques may be applied to the crosspoint cache architecture is presented in Chapter 6. By combining these two approaches, a substantial increase in maximum feasible size of shared memory multiprocessor systems is possible.

CHAPTER 2

BACKGROUND

2.1 Cache memories

One of the most effective solutions to the bandwidth problem of multis is to associate a cache memory with each CPU. A cache is a buffer memory used to temporarily hold copies of portions of main memory that are currently in use. A cache memory significantly reduces the main memory traffic for each processor, since most memory references are handled in the cache.

2.1.1 Basic cache memory architecture

The simplest cache memory arrangement is called a *direct mapped* cache. Figure 2.1 shows the design of this type of cache memory and its associated control logic. The basic unit of data in a cache is called a *line* (also sometimes called a *block*). All lines in a cache are the same size, and this size is determined by the particular cache hardware design. In current machines, the line size is always either the basic word size of the machine or the product of the word size and a small integral power of two. For example, most current processors have a 32 bit (4 byte) word size. For these processors, cache line sizes of 4, 8, 16, 32, or 64 bytes would be common. Associated with each line of data is an address tag and some control information. The combination of a data line and its associated address tag and control information is called a cache entry. The cache shown in Figure 2.1 has eight entries. In practical cache designs, the number of entries is generally a power of two in the range 64 to 8192.

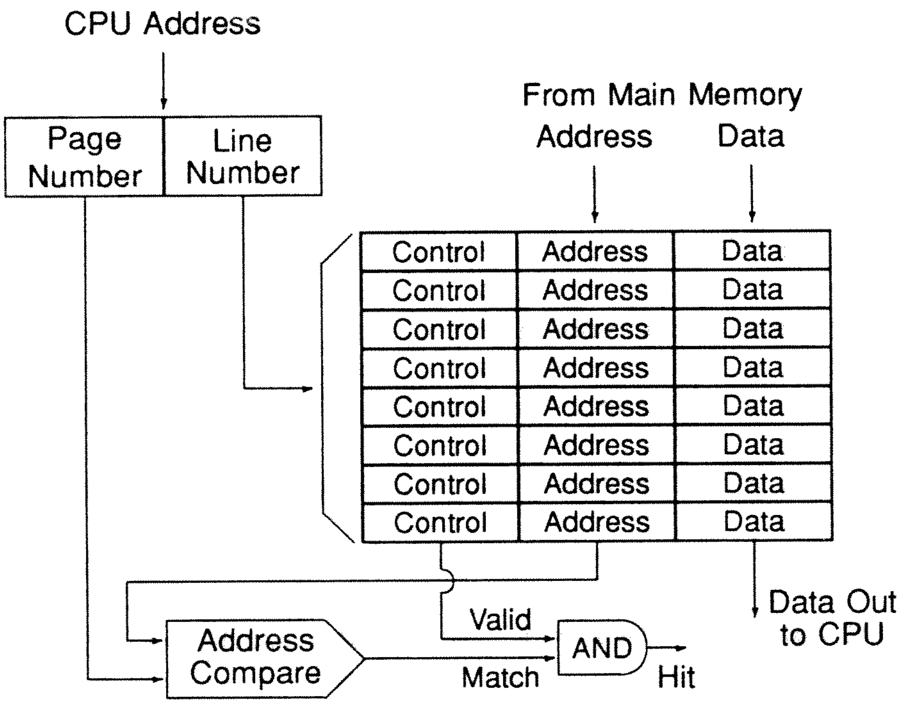


Figure 2.1: Direct mapped cache

The operation of this cache begins when an address is received from the CPU. The address is separated into a line number and a page number, with the lowest order bits forming the line number. In the example shown, only the three lowest bits would be used to form the line number, since there are only eight lines to select from. The line number is used as an address into the cache memory to select the appropriate line of data along with its address tag and control information.

The address tag from the cache is compared with the page number from the CPU address to see if the line stored in the cache is from the desired page. It is also necessary to check a bit in the control information for the line to see if it contains valid data. The data in a line may be invalid for several reasons: the line has not been used since the system was initialized, the line was invalidated by the operating system after a context switch, or the line was invalidated as part of a cache consistency protocol. If the addresses match and the line is valid, the reference is said to be a *hit*. Otherwise, the reference is classified as a *miss*.

If the CPU was performing a read operation and a hit occurred, the data from the cache is used, avoiding the bus traffic and delay that would occur if the data had to be obtained from main memory. If the CPU was performing a write operation and a hit occurred, bus usage is dependent on the cache design. The two general approaches to handling write operations are *write through* (also called *store through*) and *write back* (also called *copy back*, *store back*, or *write to*). In a write through cache, when a write operation modifies a line in the cache, the new data is also immediately transmitted to main memory. In a write back cache, write operations affect only the cache, and main memory is updated later when the line is removed from the cache. This typically occurs when the line must be replaced by a new line from a different main memory address.

When a miss occurs, the desired data must be read from or written to main memory using the system bus. The appropriate cache line must also be loaded, along with its corresponding address tag. If a write back cache is being used, it is necessary to determine whether bringing a new line into the cache will replace a line that is valid and has been modified since it was loaded from main memory. Such a line is said to be *dirty*. Dirty lines are identified by keeping a bit in the control information associated with the line that is set when the line is written to and cleared when a new line is loaded from main memory. This bit is called a *dirty bit*. The logic used to control the transfer of lines between the cache and main memory is not shown in detail in Figure 2.1.

The design shown in Figure 2.1 is called a *direct mapped* cache, since each line in main memory has only a single location in the cache into which it may be placed. A disadvantage of this design is that if two or more frequently referenced locations in main memory map to the same location in the cache, only one of them can ever be in the cache at any given time. To overcome this limitation, a design called a *set associative* cache may be used. In a two way set associative cache, the entire memory array and its associated address comparator logic is replicated twice. When an address is obtained from the CPU, both halves are checked simultaneously for a possible hit. The advantage of this scheme is that each line in main memory now has two possible cache locations instead of one. The disadvantages are that two sets of address comparison logic are needed and additional logic is needed to determine which half to load a new line into when a miss occurs. In commercially available machines, the degree of set associativity has always been a power of two ranging from one (direct mapped) to sixteen. A cache which allows a line from main memory to be placed in any location in the cache is called a *fully associative* cache. Although this design completely eliminates the problem of having multiple memory lines map to the same cache location, it requires an address comparator for every line in the cache. This makes it impractical to build large fully associative caches, although advances in VLSI technology may eventually permit their construction.

Almost all modern mainframe computers, and many smaller machines, use cache memories to improve performance. Cache memories improve performance because they have much shorter access times than main memories, typically by a factor of four to ten. Two factors contribute to their speed. Since cache memories are much smaller than main memory, it is practical to use a very fast memory technology such as ECL (emitter coupled logic) RAM. Cost and heat dissipation limitations usually force the use of a slower technology such as MOS dynamic RAM for main memory. Cache memories also can have closer physical and logical proximity to the processor since they are smaller and are normally accessed by only a single processor, while main memory must be accessible to all processors in a multi.

2.1.2 Cache operation

The successful operation of a cache memory depends on the locality of memory references. Over short periods of time, the memory references of a program will be distributed nonuniformly over its address space, and the portions of the address space which are referenced most frequently tend to remain the same over long periods of time. Several factors contribute to this locality: most instructions are executed sequentially, programs spend much of their time in loops, and related data items are frequently stored near each other. Locality can be characterized by two properties. The first, reuse or *temporal locality*, refers to the fact that a substantial fraction of locations referenced in the near future will have been referenced in the recent past. The second, prefetch or *spatial locality*, refers to the fact that a substantial fraction of locations referenced in the near future will be to locations near recent past references. Caches exploit temporal locality by saving recently referenced data so it can be rapidly accessed for future reuse. They can take advantage of spatial locality by prefetching information lines consisting of the contents of several contiguous memory locations.

Several of the cache design parameters will have a significant effect on system performance. The choice of line size is important. Small lines have several advantages:

- They require less time to transmit between main memory and cache.
- They are less likely to contain unneeded information.
- They require fewer memory cycles to access if the main memory width is narrow.

On the other hand, large lines also have advantages:

- They require fewer address tag bits in the cache.
- They reduce the number of fetch operations if all the information in the line is actually needed (prefetch).
- Acceptable performance is attainable with a lower degree of set associativity. (This is not intuitively obvious; however, results in [Smith82] support this.)

Since the unit of transfer between the cache and main memory is one line, a line size of less than the bus width could not use the full bus width. Thus it definitely does not make sense to have a line size smaller than the bus width.

The treatment of memory write operations by the cache is also of major importance here. Write back almost always requires less bus bandwidth than write through, and since bus bandwidth is such a critical performance bottleneck in a multi, it is almost always a mistake to use a write through cache.

Two cache performance parameters are of particular significance in a multi. The *miss ratio* is defined as the number of cache misses divided by the number of cache accesses. It is the probability that a referenced line is not in the cache. The *traffic ratio* is defined as the ratio of bus traffic in a system with a cache memory to that of the same system without the cache. Both the miss ratio and the traffic ratio should be as low as possible. If the CPU word size and the bus width are equal, and a write through cache with a line size of one word is used, then the miss ratio and the traffic ratio will be equal, since each miss will result in exactly one bus cycle. In other cases, the miss and traffic ratios will generally be different. If the cache line size is larger than the bus width, then each miss will require multiple bus cycles to bring in a new line. If a write back cache is used, additional bus cycles will be needed when dirty lines must be written back to the cache.

Selecting the degree of set associativity is another important tradeoff in cache design. For a given cache size, the higher the degree of set associativity, the lower the miss ratio. However, increasing the degree of set associativity increases the cost and complexity of a cache, since the number of address comparators needed is equal to the degree of set associativity. Recent cache memory research has produced the interesting result that a direct mapped cache will often outperform a set associative (or fully associative) cache of the same size even though the direct mapped cache will have a higher miss ratio. This is because the increased complexity of set associative caches significantly increases the access time for a cache hit. As cache sizes become larger, a reduced access time for hits becomes more important than the small reduction in miss ratio that is achieved through associativity. Recent studies using trace driven simulation methods have demonstrated that direct mapped caches have significant performance advantages over set associative caches for cache sizes of 32K bytes and larger [Hill87, Hill88].

2.1.3 Previous cache memory research

[Smith82] is an excellent survey paper on cache memories. Various design features and tradeoffs of cache memories are discussed in detail. Trace driven simulations are used to provide realistic performance estimates for various implementations. Specific aspects that are investigated include: line size, cache size, write through versus write back, the behavior of split data/instruction caches, the effect of input/output through the cache, the fetch algorithm, the placement and replacement algorithms, and multicache consistency. Translation lookaside buffers are also considered. Examples from real machines are used throughout the paper.

[SG83] discusses architectures for instruction caches. The conclusions are supported with experimental results using instruction trace data. [PGHLNSV83] describes the architecture of an instruction cache for a RISC (Reduced Instruction Set Computer) processor.

[HS84] provides extensive trace driven simulation results to evaluate the performance of cache memories suitable for on-chip implementation in microprocessors. [MR85] discusses cache performance in Motorola MC68020 based systems.

2.1.4 Cache consistency

A problem with cache memories in multiprocessor systems is that modifications to data in one cache are not necessarily reflected in all caches, so it may be possible for a processor to reference data that is not current. Such data is called *stale data*, and this problem is called the *cache consistency* or *cache coherence* problem. A general discussion of this problem is presented in the [Smith82] survey paper. This is a serious problem for which no completely satisfactory solution has been found, although considerable research in this area has been performed.

The standard software solution to the cache consistency problem is to place all shared writable data in non-cacheable storage and to flush a processor's cache each time the processor performs a context switch. Since shared writable data is non-cacheable, it cannot become inconsistent in any cache. Unshared data could potentially become inconsistent if a process migrates from one processor to another; however, the cache flush on context switch prevents this situation from occurring. Although this scheme does provide consistency, it does so at a very high cost to performance.

The classical hardware solution to the cache consistency problem is to broadcast all writes. Each cache sends the address of the modified line to all other caches. The other caches invalidate the modified line if they have it. Although this scheme is simple to implement, it is not practical unless the number of processors is very small. As the number of processors is increased, the cache traffic resulting from the broadcasts rapidly becomes prohibitive.

An alternative approach is to use a centralized directory that records the location or locations of each line in the system. Although it is better than the broadcast scheme, since it avoids interfering with the cache accesses of other processors, directory access conflicts can become a bottleneck.

The most practical solutions to the cache consistency problem in a system with a large number of processors use variations on the directory scheme in which the directory information is distributed among the caches. These schemes make it possible to construct systems in which the only limit on the maximum number of processors is that imposed by the total bus and memory bandwidth. They are called "snooping cache" schemes [KEWPS85], since each cache must monitor addresses on the system bus, checking each reference for a possible cache hit. They have also been referred to as "two-bit directory" schemes [AB84], since each line in the cache usually has two bits associated with it to specify one of four states for the data in the line.

[Goodm83] describes the use of a cache memory to reduce bus traffic and presents a description of the *write-once* cache policy, a simple snooping cache scheme. The write-once scheme takes advantage of the broadcast capability of the shared bus between the local caches and the global main memory to dynamically classify cached data as local or shared, thus ensuring cache consistency without broadcasting every write operation or using a global directory. Goodman defines the four cache line states as follows: 1) **Invalid**, there is no data in the line; 2) **Valid**, there is data in the line which has been read from main memory and has not been modified (this is the state which always results after a read miss has been serviced); 3) **Reserved**, the data in the line has been locally modified exactly once since it has been brought into the cache and the change has been written through to main memory; and 4) **Dirty**, the data in the line has been locally modified more than once since it was brought into the cache and the latest change has not been transmitted to main memory.

Since this is a snooping cache scheme, each cache must monitor the system bus and check all bus references for hits. If a hit occurs on a bus write operation, the appropriate line in the cache is marked invalid. If a hit occurs on a read operation, no action is taken unless the state of the line is reserved or dirty, in which case its state is changed to valid. If the line was dirty, the cache must inhibit the read operation on main memory and supply the data itself. This data is transmitted to both the cache making the request and main memory. The design of the protocol ensures that no more than one copy of a particular line can be dirty at any one time.

The need for access to the cache address tags by both the local processor and the system bus makes these tags a potential bottleneck. To ease this problem, two identical copies of the tag memory can be kept, one for the local processor and one for the system bus. Since the tags are read much more often than they are written, this allows the processor and bus to access them simultaneously in most cases. An alternative would be to use dual ported memory for the tags, although currently available dual ported memories are either too expensive, too slow, or both to make this approach very attractive. Goodman used simulation to investigate the performance of the write-once scheme. In terms of bus traffic, it was found to perform about as well as write back and it was superior to write through.

[PP84] describes another snooping cache scheme. The states are named **Invalid**, **Exclusive-Unmodified**, **Shared-Unmodified**, and **Exclusive-Modified**, corresponding respectively to **Invalid**, **Reserved**, **Valid**, and **Dirty** in [Goodm83]. The scheme is nearly identical to the write-once scheme, except that when a line is loaded following a read miss, its state is set to Exclusive-Unmodified if the line was obtained from main memory, and it is set to Shared-Unmodified if the line was obtained from another cache, while in the write-once scheme the state would be set to Valid (Shared-Unmodified) regardless of where the data is obtained. [PP84] notes that the change reduces unnecessary bus traffic when a line is written after it is read. An approximate analysis was used to estimate the performance of this scheme, and it appears to perform well as long as the fraction of data that is shared between processors is small.

[RS84] describes two additional versions of snooping cache schemes. The first, called the **RB** scheme (for "read broadcast"), has only three states, called **Invalid**, **Read**, and **Local**. The read and local states are similar to the valid and dirty states, respectively, in the write-once scheme of [Goodm83], while there is no state corresponding to the reserved state (a "dirty" state is assumed immediately after the first write). The second, called the **RWB** scheme (presumably for "read write broadcast"), adds a fourth state called **First** which corresponds to the reserved state in write-once. A feature of RWB not present in write-once is that when a cache detects that a line read from main memory by another processor will hit on an invalid line, the data is loaded into the invalid line on the grounds that it might be used, while the invalid line will certainly not be useful. The advantages of this are debatable, since loading the line will tie up cache cycles that might be used by the processor on that cache, and the probability of the line being used may be low. [RS84] is concerned primarily with formal correctness proofs of these schemes and does not consider the performance implications of practical implementations of them.

[AB84] discusses various solutions to the cache consistency problem, including broadcast, global directory, and snooping approaches. Emphasis is on a snooping approach in which the states are called **Absent**, **Present1**, **Present***, and **PresentM**. This scheme is generally similar to that of [PP84], except that two-bit tags are associated with lines in main memory as well as with lines in caches. An approximate analysis of this scheme is used to estimate the maximum useful number of processors for various situations. It is shown that if the level of data sharing is reasonably low, acceptable performance can be obtained for as many as 64 processors.

[KEWPS85] describes the design and VLSI implementation of a snooping cache scheme, with the restriction that the design be compatible with current memory and backplane designs. This scheme is called the Berkeley Ownership Protocol, with states named **Invalid**, **UnOwned**, **Owned Exclusively**, and **Owned NonExclusively**. Its operation is quite similar to that of the scheme described in [PP84]. [KEWPS85] suggests having the compiler include in its generated code indications of which data references are likely to be to non-shared read/write data. This information is used to allow the cache controller to obtain exclusive access to such data in a single bus cycle, saving one bus cycle over the scheme in which the data is first obtained as shared and then as exclusive.

2.1.5 Performance of cache consistency mechanisms

Although the snooping cache approaches appear to be similar to broadcasting writes, their performance is much better. Since the caches record the shared or exclusive status of each line, it is only necessary to broadcast writes to shared lines on the bus; bus activity for exclusive lines is avoided. Thus, the cache bandwidth problem is much less severe than for the broadcast writes scheme.

The protocols for enforcing cache consistency with snooping caches can be divided into two major classes. Both use the snooping hardware to dynamically identify shared writable lines, but they differ in the way in which write operations to shared lines are handled.

In the first class of protocols, when a processor writes to a shared line, the address of the line is broadcast on the bus to all other caches, which then invalidate the line. Two examples are the Illinois protocol and the Berkeley Ownership Protocol [PP84, KEWPS85]. Protocols in this class are called *write-invalidate* protocols.

In the second class of protocols, when a processor writes to a shared line, the written data is broadcast on the bus to all other caches, which then update their copies of the line. Cache invalidations are never performed by the cache consistency protocol. Two examples are the protocol in DEC's Firefly multiprocessor workstation and that in the Xerox Dragon multiprocessor [TS87, AM87]. Protocols in this class are called *write-broadcast* protocols.

Each of these two classes of protocol has certain advantages and disadvantages, depending on the pattern of references to the shared data. For a shared data line that tends to be read and written several times in succession by a single processor before a different processor references the same line, the write-invalidate protocols perform better than the write-broadcast protocols. The write-invalidate protocols use the bus to invalidate the other copies of a shared line each time a new processor makes its first reference to that shared line, and then no further bus accesses are necessary until a different processor accesses that line. Invalidation can be performed in a single bus cycle, since only the address of the modified line must be transmitted. The write-broadcast protocols, on the other hand, must use the bus for every write operation to the shared data, even when a single processor writes to the data several times consecutively. Furthermore, multiple bus cycles may be needed for the write, since both an address and data must be transmitted.

For a shared data line that tends to be read much more than it is written, with writes occurring from random processors, the write-broadcast protocols tend to perform better than the write-invalidate protocols. The write-broadcast protocols use a single bus operation (which may involve multiple bus cycles) to update all cached copies of the line, and all read operations can be handled directly from the caches with no bus traffic. The write-invalidate protocols, on the other hand, will invalidate all copies of the line each time it is written, so subsequent cache reads from other processors will miss until they have reloaded the line.

A comparison of several cache consistency protocols using a simulation model is described in [AB86]. This study concluded that the write-broadcast protocols gave superior performance. A limitation of this model is the assumption that the originating processors for a sequence of references to a particular line are independent and random. This strongly biases the model against write-invalidate protocols. Actual parallel programs are likely to have a less random sequence of references; thus, the model may not be a good reflection of reality.

A more recent comparison of protocols is presented in [VLZ88]. In this study, an analytical performance model is used. The results show less difference in performance between write-broadcast and write-invalidate protocols than was indicated in [AB86]. However, as in [AB86], the issue of processor locality in the sequence of references to a particular shared block is not addressed. Thus, there is insufficient information to judge the applicability of this model to workloads in which such locality is present.

The issue of locality of reference to a particular shared line is considered in detail in [EK88]. This paper also discusses the phenomenon of *passive sharing* which can cause significant inefficiency in write-broadcast protocols. Passive sharing occurs when shared lines that were once accessed by a processor but are no longer being referenced by that processor remain in the processor's cache. Since this line will remain identified as shared, writes to the line by another processor must be broadcast on the bus, needlessly wasting bus bandwidth. Passive sharing is more of a problem with large caches than with small ones, since a large cache is more likely to hold inactive lines for long intervals. As advances in memory technology increase practical cache sizes, passive sharing will become an increasingly significant disadvantage of write-broadcast protocols.

Another concept introduced in this paper is the *write run*, which is a sequence of write references to a shared line by a single processor, without interruption by accesses of any kind to that line from other processors. It is demonstrated that in a workload with short write runs, write-broadcast protocols provide the best performance, while when the average write run length is long, write-invalidate protocols will be better. This result is expected from the operation of the protocols. With write-broadcast protocols, every write operation causes a bus operation, but no extra bus operations are necessary when active accesses to a line move from one processor to another. With write-invalidate protocols, bus operations are only necessary when active accesses to a line move from one processor to another. The relation between the frequency of writes to a line and the frequency with which accesses to the line move to a different processor is expressed in the length of the write run. With short write runs, accesses to a line frequently move to a different processor, so the write-invalidate protocols produce a large number of invalidations that are unnecessary with the write-broadcast protocols. On the other hand, with long write runs, a line tends to be written many times in succession by a single processor, so the write-broadcast protocols produce a large number of bus write operations that are unnecessary with the write-invalidate protocols.

Four parallel application workloads were investigated. It was found that for two of them, the average write run length was only 2.09 and a write-broadcast protocol provided the best performance, while for the other two, the average write run length was 6.0 and a write-invalidate protocol provided the best performance.

An adaptive protocol that attempts to incorporate some of the best features of each of the two classes of cache consistency schemes is proposed in [Archi88]. This protocol, called EDWP (Efficient Distributed-Write Protocol), is essentially a write broadcast protocol with the following modification: if some processor issues three writes to a shared line with no intervening references by any other processors, then all the other cached copies of that line are invalidated and the processor that issued the writes is given exclusive access to the line. This eliminates the passive sharing problem. The particular number of successive writes allowed to occur before invalidating the line (the length of the write run), three, was selected based on a simulated workload model. A simulation model showed that EDWP performed better than write-broadcast protocols for some workloads, and the performance was about the same for other workloads. A detailed comparison with write-invalidate protocols was not presented, but based on the results in [EK88], the EDWP protocol can be expected to perform significantly better than write-invalidate protocols for short average write run lengths, while performing only slightly worse for long average write run lengths.

The major limitation of all of the snooping cache schemes is that they require all processors to share a common bus. The bandwidth of a single bus is typically insufficient for even a few dozen processors. Higher bandwidth interconnection networks such as crossbars and multistage networks cannot be used with snooping cache schemes, since there is no simple way for every cache to monitor the memory references of all the other processors.

2.2 Maximizing single bus bandwidth

Although cache memories can produce a dramatic reduction in bus bandwidth requirements, bus bandwidth still tends to place a serious limitation on the maximum number of processors in a multi. [Borri85] presents a detailed discussion of current standard implementations of 32 bit buses. It is apparent that the bandwidth of these buses is insufficient to construct a multi with a large number of processors. Many techniques have been used for maximizing the bandwidth of a single bus. These techniques can be grouped into the following categories:

- Minimize bus cycle time
- Increase bus width
- Improve bus protocol

2.2.1 Minimizing bus cycle time

The most straightforward approach for increasing bus bandwidth is to make the bus very fast. While this is generally a good idea, there are limitations to this approach.

Interface logic speed and propagation delay considerations place an upper bound on the bus speed. These factors are analyzed in detail in Chapter 3 of this dissertation.

2.2.2 Increasing bus width

To allow a larger number of processors to be used while avoiding the problems inherent with multiple buses, a single bus with a wide datapath can be used. We propose the term *fat bus* for such a bus.

The fat bus has several advantages over multiple buses. It requires fewer total signals for a given number of data signals. For example, a 32 bit bus might require approximately 40 address and control signals for a total of 72 signals. A two word fat bus would have 64 data signals but would still need only 40 address and control signals, so the total number of signals is 104. On the other hand, using two single word buses would double the total number of signals from 72 to 144. Another advantage is that the arbitration logic for a single fat bus is simpler than that for two single word buses.

An upper limit on bus width is imposed by the cache line size. Since the cache will exchange data with main memory one line at a time, a bus width greater than the line size is wasteful and will not improve performance. The cache line size is generally limited by the cache size; if the size of a cache line is too large compared with the total size of the cache, the cache will contain too few lines, and the miss ratio will degrade as a result. A detailed study of the tradeoffs involved in selecting the cache line size is presented in [Smith87b].

2.2.3 Improving bus protocol

In the simplest bus design for a multi, a memory read operation is performed as follows: the processor uses the arbitration logic to obtain the use of the bus, it places the address on the bus, the addressed memory module places the data on the bus, and the processor releases the bus.

This scheme may be modified to decouple the address transmission from the data transmission. When this is done, a processor initiates a memory read by obtaining the use of the bus, placing the address on the bus, and releasing the bus. Later, after the memory module has obtained the data, the memory module obtains the use of the bus, places both the address and the data on the bus, and then releases the bus. This scheme is sometimes referred to as a time shared bus or a split transaction bus. Its advantage is that additional bus transactions may take place during the memory access time. The disadvantage is that two bus arbitration operations are necessary. Furthermore, the processors need address comparator logic in their bus interfaces to determine when the data they have requested has become available. It is not reasonable to use this technique unless the bus arbitration time is significantly less than the memory access time.

Another modification is to allow a burst of data words to be sent in response to a single address. This approach is sometimes called a packet bus. It is only useful in situations in which a single operation references multiple contiguous words. Two instances of this are: fetching a cache line when the line size is greater than the bus width, and performing an operation on a long operand such as an extended precision floating point number.

2.3 Multiple bus architecture

One solution to the bandwidth limitation of a single bus is to simply add additional buses. Consider the architecture shown in Figure 2.2 that contains N processors, $\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_N$, each having its own private cache, and all connected to a shared memory by B buses $\mathrm{B}_1, \mathrm{B}_2, \ldots, \mathrm{B}_B$. The shared memory consists of M interleaved banks $\mathrm{M}_1, \mathrm{M}_2, \ldots, \mathrm{M}_M$ to allow simultaneous memory requests concurrent access to the shared memory. This avoids the loss in performance that occurs if those accesses must be serialized, which is the case when there is only one memory bank. Each processor is connected to every bus and so is each memory bank. When a processor needs to access a particular bank, it has B buses from which to choose. Thus each processor-memory pair is connected by several redundant paths, which implies that the failure of one or more paths can, in principle, be tolerated at the cost of some degradation in system performance.

In a multiple bus system several processors may attempt to access the shared memory simultaneously. To deal with this, a policy must be implemented that allocates the available buses to the processors making requests to memory. In particular, the policy must deal with the case when the number of processors exceeds B. For performance reasons this allocation must be carried out by hardware arbiters which, as we shall see, add significantly to the complexity of the multiple bus interconnection network.

There are two sources of conflict due to memory requests in the system of Figure 2.2. First, more than one request can be made to the same memory module, and, second, there may be an insufficient bus capacity available to accommodate all the requests.

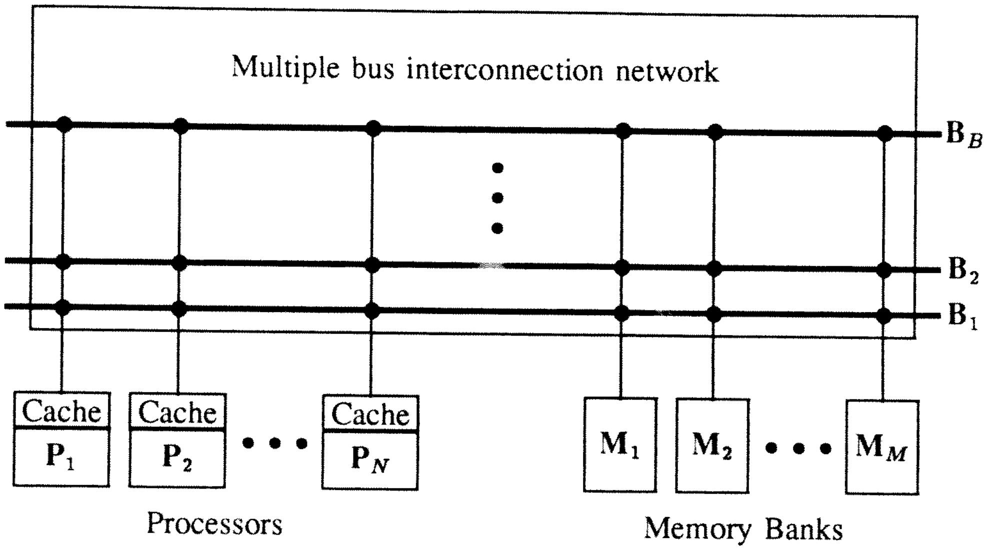


Figure 2.2: Multiple bus multiprocessor

Correspondingly, the allocation of a bus to a processor that makes a memory request requires a two-stage process as follows:

1. Memory conflicts are resolved first by M 1-of-N arbiters, one per memory bank. Each 1-of-N arbiter selects one request from up to N requests to get access to the memory bank.

2. Memory requests that are selected by the memory arbiters are then allocated a bus by a B-of-M arbiter. The B-of-M arbiter selects up to B requests from one or more of the M memory arbiters.

The assumption that the address and data paths operate asynchronously allows arbitration to be overlapped with data transfers.

2.3.1 Multiple bus arbiter design

As we have seen, a general multiple bus system calls for two types of arbiters: 1-of-N arbiters to select among processors and a B-of-M arbiter to allocate buses to those processors that were successful in obtaining access to memory.

1-of-N arbiter design

If multiple processors require exclusive use of a shared memory bank and access it on an asynchronous basis, conflicts may occur. These conflicts can be resolved by a 1-of-N arbiter. The signaling convention between the processors and the arbiter is as follows: Each processor P_i has a request line R_i and a grant line G_i. Processor P_i requests a memory access by activating R_i and the arbiter indicates the allocation of the requested memory bank to P_i by activating G_i.

Several designs for 1-of-N arbiters have been published [PFL75]. In general, these designs can be grouped into three categories: fixed priority schemes, rings, and trees. Fixed priority arbiters are relatively simple and fast, but they have the disadvantage that they are not fair in that lower priority processors can be forced to wait indefinitely if higher priority processors keep the memory busy. A ring structured arbiter gives priority to the processors on a rotating round-robin basis, with the lowest priority given to the processor which most recently used the memory bank being requested. This has the advantage of being fair, because it guarantees that all processors will access memory in a finite amount of time, but the arbitration time grows linearly with the number of processors. A tree structured 1-of-N arbiter is generally a binary tree of depth $\log_2 N$ constructed from 1-of-2 arbiter modules (see Figure 2.3). Each 1-of-2 arbiter module in the tree has two request input and two grant output lines, and a cascaded request output and a cascaded grant input for connection to the next arbitration stage. Tree structured arbiters are faster than ring arbiters since the arbitration time grows only as $O(\log_2 N)$ instead of $O(N)$. Fairness can be assumed by placing a flip-flop in each 1-of-2 arbiter which is toggled automatically to alternate priorities when the arbiter receives simultaneous requests.

An implementation of a 1-of-2 arbiter module constructed from 12 gates is given in [PFL75]. The delay from the request inputs to the cascaded request output is 2Δ, where Δ denotes the nominal gate delay, and the delay from the cascaded grant input to the grant outputs is Δ. Thus, the total delay for a 1-of-N arbiter tree is $3\Delta \log_2 N$. So, for example, to construct a 1-of-64 arbiter, a six-level tree is needed. This tree will contain 63 1-of-2 arbiters, for a total of 756 gates. The corresponding total delay imposed by the arbiter will be 18Δ.

B-of-*M* arbiter design

Detailed implementations of B-of-M arbiters are given in [LV82]. The basic arbiter consists of an iterative ring of M arbiter modules $\mathrm{A}_1, \mathrm{A}_2, \ldots \mathrm{A}_M$ that compute the bus assignments, and a state register to store the arbiter state after each arbitration cycle (see Figure 2.4). The storage of the state is necessary to make the arbiter fair by taking into account previous bus assignments. After each arbitration cycle, the highest priority is given to the module just after the last one serviced. This is a standard round-robin policy.

An arbitration cycle starts with all of the buses marked as available. The state

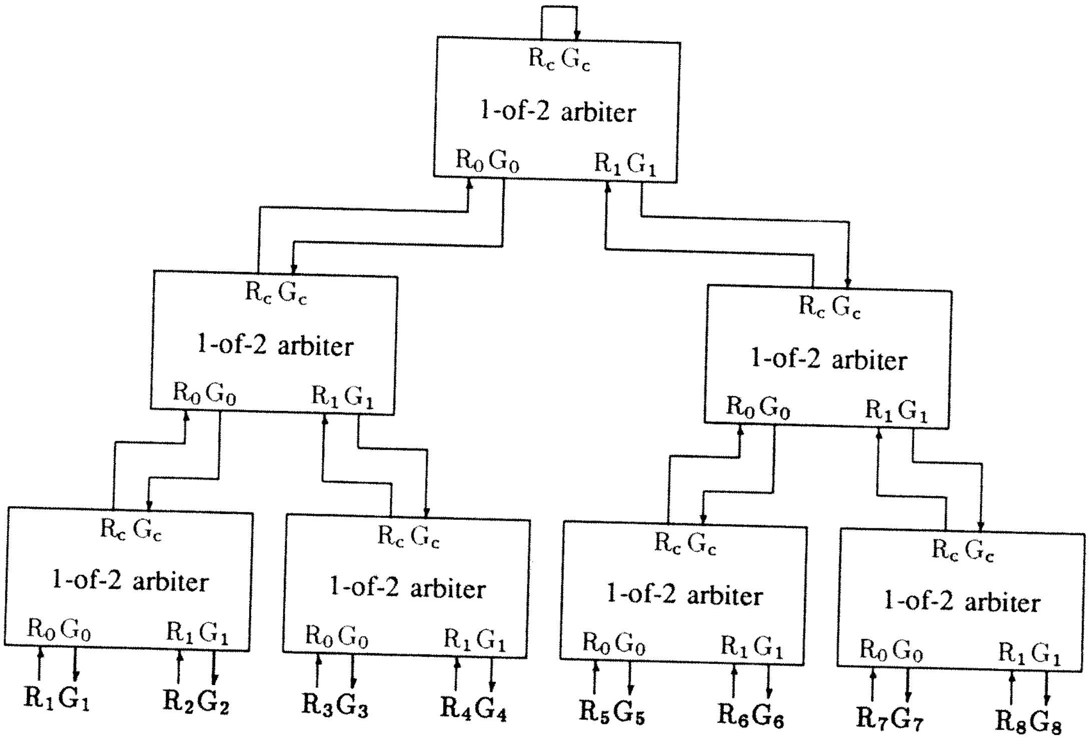


Figure 2.3: 1-of-8 arbiter constructed from a tree of 1-of-2 arbiters

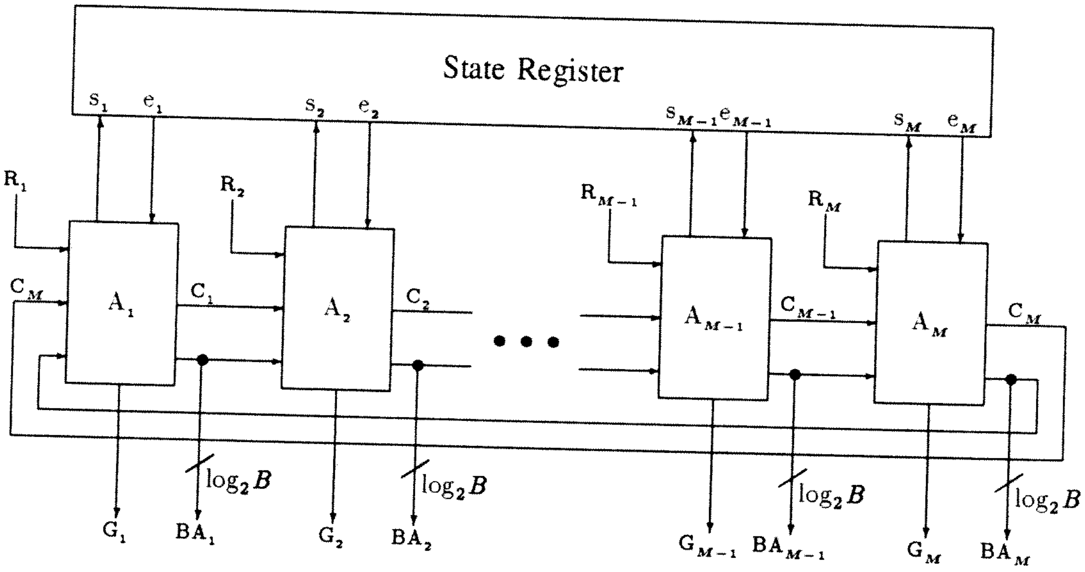


Figure 2.4: Iterative design for a B-of-M arbiter

register identifies the highest priority arbiter module, A_i, by asserting signal e_i to that module. Arbitration begins with this module and proceeds around the ring from left to right. At each arbiter module, the R_i input is examined to see if the corresponding memory bank M_i is requesting a bus. If a request is present and a bus is available, the address of the first available bus is placed on the BA_i output and the G_i signal is asserted. BA_i is also passed to the next module, to indicate the highest numbered bus that has been assigned. If a module does not grant a bus, its BA_i output is equal to its BA_{i-1} input. If a module does grant a bus, its BA_i output is set to $\mathrm{BA}_{i-1} + 1$. When $\mathrm{BA}_i = B$ all the buses have been used and the assignment process stops. The highest priority module, as indicated by the e_i signal, ignores its BA_i input and begins bus assignment with the first bus by setting $\mathrm{BA}_i = 1$. Each module's C_i input is a signal from the previous module which indicates that the previous module has completed its bus assignment. Arbitration proceeds sequentially through the modules until all of the buses have been assigned, or all the requests have been satisfied. The last module to assign a bus asserts its s_i signal. This is recorded in the state register, which uses it to select the next e_i output so that the next arbitration cycle will begin with the module immediately after the one that assigned the last bus.

Turning to the performance of B-of-M arbiters, we observe that the simple iterative design of Figure 2.4 must have a delay proportional to M, the number of arbiter modules. By combining g of these modules into a single module (the "lookahead" design of [LV82]), the delay is reduced by a factor of g. If the enlarged modules are implemented by PLAs with a delay of 3Δ, the resulting delay of the arbiter is about $\frac{3M}{g}\Delta$. For example, where $M = 16$ and $g = 4$, the arbiter delay is about 12Δ.

If the lookahead design approach of [LV82] is followed, the arbitration time of B-of-M arbiters grows at a rate greater than $O(\log_2 M)$ but less than $O(\log_2{}^2 M)$, so the delay of the B-of-M arbiter could become the dominant performance limitation for large M.

2.3.2 Multiple bus performance models

Many analytic performance models of multiple bus and crossbar systems have been published [Strec70, Bhand75, BS76, Hooge77, LVA82, GA84, MHBW84, Humou85, Towsl86]. The major problem with these studies is the lack of data to validate the models developed. Although most of these studies compared their models with the results of simulations, all of the simulations except for those in [Hooge77] used memory reference patterns derived from random number generators and not from actual programs. The traces used in [Hooge77] consisted of only 10,000 memory references, which is extremely small by current standards. For example, with a 128 Kbyte cache and a word size of four bytes, at least 32,768 memory references are needed just to fill the cache.

2.3.3 Problems with multiple buses

The multiple bus approach has not seen much use in practical systems. The major reasons for this include difficulties with cache consistency, synchronization, and arbitration.

It is difficult to implement hardware cache consistency in a multiple bus system. The principal problem is that each cache needs to monitor every cycle on every bus.

This would be impractical for more than a few buses, since it would require extremely high bandwidth for the cache address tags.

Multiple buses can also cause problems with serializability. If two processors reference the same line (using two different buses), they could each modify a copy of the line in the other's cache, thus leaving that line in an inconsistent state.

Finally, the arbitration logic required for a multiple bus system is very complex. The complexity of assigning B buses to P processors grows rapidly as B and P increase. As a result of this, the arbitration circuitry will introduce substantial delays unless the number of buses and processors is very small.

2.4 Summary

Cache memories are a critical component of modern high performance computer systems, especially multiprocessor systems. When cache memories are used in a multiprocessor system, it is necessary to prevent data from being modified in multiple caches in an inconsistent manner. Efficient means for ensuring cache consistency require a shared bus, so that each cache can monitor the memory references of the other caches.

The limited bandwidth of the shared bus can impose a substantial performance limitation in a single bus multiprocessor. Solutions to this bandwidth problem are investigated in Chapters 3 and 4.

Multiple buses can be used to obtain higher total bandwidth, but they introduce difficult cache consistency and bus arbitration problems. A modified multiple bus architecture that avoids these problems is described in detail in Chapter 5 of this dissertation.

CHAPTER 3

BUS PERFORMANCE MODELS

3.1 Introduction

The maximum rate at which data can be transferred over a bus is called the *bandwidth* of the bus. The bandwidth is usually expressed in bytes or words per second. Since all processors in a multi must access main memory through the bus, its bandwidth tends to limit the maximum number of processors.

The low cost dynamic RAMs that would probably be used for the main memory have a bandwidth limitation imposed by their cycle time, so this places an additional upper bound on system performance.

To illustrate these limitations, consider a system built with Motorola MC68020 microprocessors. For the purposes of this discussion, a word will be defined as 32 bits. A 16.67 MHz 68020 microprocessor accesses memory at a rate of approximately 2.8 million words per second [MR85].

With 32-bit wide memory, to provide adequate bandwidth for N 16.67 MHz 68020 processors, a memory cycle time of $\frac{357}{N}$ ns or less is needed (357 ns is the reciprocal of 2.8 million words per second, the average memory request rate). The fastest dynamic RAMs currently available in large volume have best case access times of approximately 50 ns [Motor88] (this is for static column RAMS, assuming a high hit rate within the current column). Even with this memory speed, a maximum of only 7 processors could be supported without saturating the main memory. In order to obtain a sufficient data rate from main memory, it is often necessary to divide the main memory into several modules. Each module is controlled independently and asynchronously from the other modules. This technique is called *interleaving*. Without interleaving, memory requests can only be serviced one at a time, since each request must finish using the memory before the next request can be sent to the memory. With interleaving, there can be one outstanding request per module. By interleaving the main memory into a sufficient number of modules, the main memory bandwidth problem can be overcome.

Bus bandwidth imposes a more serious limitation. The VME bus (IEEE P1014), a typical 32-bit bus, can supply 3.9 million words per second if the memory access time is 100 nanoseconds, while the Fastbus (ANSI/IEEE 960), a high performance 32-bit bus, can supply 4.8 million words per second from 100 ns memory [Borri85]. Slower memory will decrease these rates. From this information, it is clear that the bandwidth of either of these buses is inadequate to support even two 68020 processors without slowing down the processors significantly. Furthermore, this calculation does not even consider the time required for bus arbitration, which is at least 150 ns for the VME bus and 90 ns for the Fastbus. These buses are obviously not suitable for the interconnection network in a high performance shared memory multiprocessor system unless cache memories are used to significantly reduce the bus traffic per processor.

To ease the problem of limited bus and memory bandwidth, cache memories may be used. By servicing most of the memory requests of a processor in the cache, the number of requests that must use the bus and main memory are greatly reduced.

The major focus of this chapter will be the bus bandwidth limitation of a single bus and specific bus architectures that may be used to overcome this limitation.

3.2 Implementing a logical single bus

To overcome the bus loading problems of a single shared bus while at the same time preserving the efficient snooping protocols possible with a single shared bus, it is necessary to construct an interconnection network that preserves the logical structure of a single bus that avoids the electrical implementation problems associated with physically attaching all of the processors directly to a single bus. There are several practical ways to construct a network that logically acts as a shared bus connecting a large number of processors. Figure 3.1 shows an implementation that uses a two-level hierarchy of buses. If a single bus can support N processors with delay Δ, then this arrangement will handle N^2 processors with delay 3Δ. Each bus shown in Figure 3.1 has delay Δ, and the worst case path is from a processor down through a level one bus, through the level two bus, and up through a different level one bus. It is necessary to consider

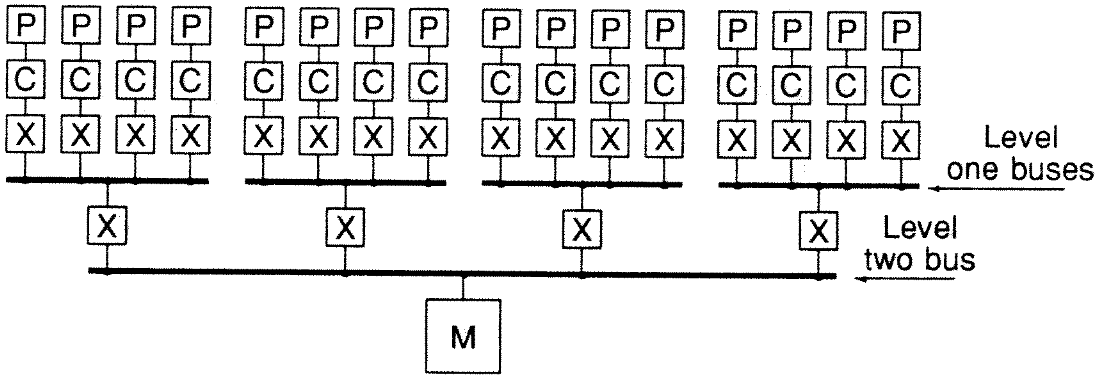


Figure 3.1: Interconnection using a two-level bus hierarchy

the worst case path between any two processors, rather than just the worst case path to main memory, since each memory request must be available to every processor to allow cache snooping.

Figure 3.2 shows another implementation consisting of a binary tree structure of transceivers. This arrangement can connect N processors (where N is a power of 2) using $2N-2$ transceivers. The maximum delay through this network is $2\log_2 N$ times the delay of a single transceiver.

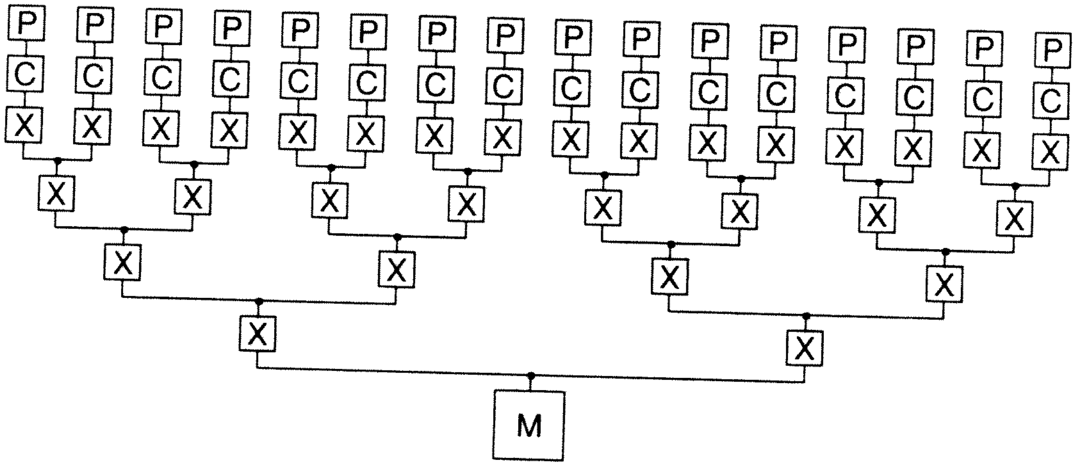


Figure 3.2: Interconnection using a binary tree

Many other network arrangements are possible. In order to select the optimal bus organization for a given implementation technology and a given number of processors, models of bus delay and bus interference are needed. In this chapter, we construct such models and use them to derive the optimal organizations and sizes for single buses, two-level bus hierarchies, and binary tree interconnections. Simple relationships are developed that express the largest useful systems that can be constructed with a given processor, cache, and bus organization. We present an example based on a TTL bus to show that shared memory multiprocessors with several dozen processors are feasible using a simple two-level bus hierarchy. We conclude our discussion on bus design with an example based on the IEEE P896 Futurebus. This example demonstrates that using a bus with better electrical characteristics can substantially increase performance.

3.3 Bus model

We define system throughput as the ratio of the total memory traffic in the system to the memory traffic of a single processor with a zero delay bus. This a useful measure of system performance since it is proportional to the total rate at which useful computations may be performed by the system for a given processor and cache design. In this section we develop a model for system throughput, T, as a function of the number of processors, N, the bus cycle time, t_c, and the mean time between shared memory requests from a processor exclusive of bus time, t_r. In other words, t_r is the sum of the mean compute time between memory references and the mean memory access time.

3.3.1 Delay model

In general, the delay associated with a bus depends on the number of devices connected to it. In this section, we will use N to represent the number of devices connected to the bus under discussion. Based on their dependence on N, the delays in a bus can be classified into four general types: constant, logarithmic, linear, and quadratic. Constant delays are independent of N. The internal propagation delay of a bus transceiver is an example of constant delay. Logarithmic delays are proportional to $\log_2 N$. The delay through the binary tree interconnection network shown in Figure 3.2 is an example of logarithmic delay. The delay of an optimized MOS driver driving a capacitive load where the total capacitance is proportional to N is another example of logarithmic delay [MC80]. Linear delays are proportional to N. The transmission line delay of a bus whose length is proportional to N is an example of linear delay. Another example is the delay of an RC circuit in which R (bus driver internal resistance) is fixed and C (bus receiver capacitance) is proportional to N. Finally, quadratic delays are proportional to N^2. The delay of an internal bus on a VLSI or WSI chip in which both the total resistance and the total capacitance of the wiring are proportional to the length of the bus is an example of quadratic delay [RJ87].

The total delay of a bus, Δ, can be modeled as the sum of these four components (some of which may be zero or negligible) as follows:

$$\Delta = k_{\text{const}} + k_{\log} \log_2 N + k_{\text{lin}} N + k_{\text{quad}} N^2$$

The minimum bus cycle time is limited by the bus delay. It is typically equal to the bus delay for a bus protocol that requires no acknowledgment, and it is equal to twice the bus delay for a protocol that does require an acknowledgment. We will assume the use of a protocol for which no acknowledgment is required. Thus, the bus cycle time t_c can be expressed as,

$$t_c = k_{\text{const}} + k_{\log} \log_2 N + k_{\text{lin}} N + k_{\text{quad}} N^2 \tag{3.1}$$

3.3.2 Interference model

To accurately model the bus performance when multiple processors share a single bus, the issue of bus interference must be considered. This occurs if two or more processors attempt to access the bus at the same time—only one can be serviced while the others must wait. Interference increases the mean time for servicing a memory request over the bus, and it causes the bus utilization for an N processor system to be less than N times that of a single processor system.

If the requests from different processors are independent, as would likely be the case when they are running separate processes in a multiprogrammed system, then a Markov chain model of bus interference can be constructed [MHBW84]. This model may be used to estimate the effective memory access time and the bus utilization. In the model, states $\Pi_0, \Pi_1, \ldots, \Pi_{N-1}$ correspond to the number of processors blocked (it is not possible to have all N processors blocked; one must be accessing memory). The corresponding steady state probabilities for these states are denoted by $\pi_0, \pi_1, \ldots, \pi_{N-1}$. The probability of a transition from one state i to another state j is denoted by $\pi_{i,j}$. Figure 3.3 illustrates the Markov chain model for $N = 4$. Since the bus can only service one request at a time, the system can move to lower states at a maximum rate of one state per bus cycle. This is indicated in the figure by a lack of transitions that go two or more states to the left; in other words, the transition probabilities $\pi_{2,0}$, $\pi_{3,1}$, and $\pi_{3,0}$ are all zero. Generalizing this to systems with arbitrary N, we have $\pi_{i,j} = 0$ if $i - j > 1$. On the other hand, since any processor that is not blocked may generate a new request in any cycle, it is possible to move to any higher numbered state in a single cycle.

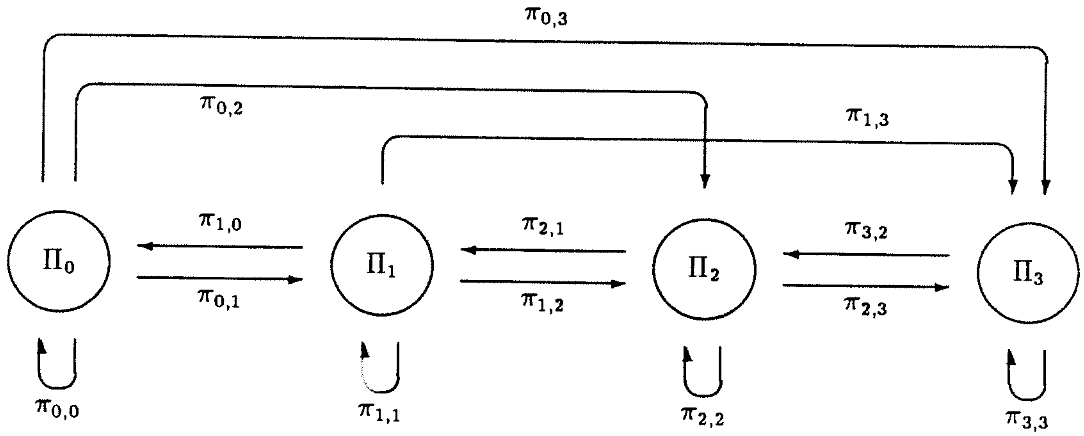


Figure 3.3: Markov chain model $(N = 4)$

The state transition probabilities $\pi_{i,j}$ in this model may be obtained from the probability that a particular number of new requests are issued. Specifically, if the system is in state Π_i, then $\pi_{i,j}$, the probability of moving to state Π_j given initial state Π_i, is equal to the conditional probability of $1 + j - i$ processors issuing new memory requests given that i processors are initially blocked. To see why this is so, consider the case when no new memory requests are issued and the initial state is Π_i. In this case, the system will move to state Π_{i-1}, since one request will be serviced. Now consider the case where k processors issue new memory requests. Each additional request will move the system to a higher state, so the new state will be Π_{k+i-1}. If we let $j = k + i - 1$, then $k = 1 + j - i$, so if $k = 1 + j - i$ processors issue new requests, the new state will be state $\Pi_j = \Pi_{k+i-1}$. To derive the conditional probability of $1 + j - i$ processors issuing new memory requests given that i processors are initially blocked, we define p as the probability that a particular processor requests a memory access in a bus cycle. For convenience, we also define q to be $1 - p$. It is assumed that p is the same for all processors and that requests are independent. This is an important simplifying assumption that we shall refer to as the *independence assumption.* Let $\alpha_{i,j}$ be the probability that j processors generate new memory requests given that i processors are blocked. Since processors that are blocked cannot generate new requests, this probability is given by the binomial distribution:

$$\alpha_{i,j} = \begin{cases} \binom{N-i}{j} p^j q^{(N-i)-j} & j \leq N-i \\ \\ 0 & j > N-i \end{cases}$$

The state transition probabilities in the Markov chain model may be easily obtained from the α values. $\pi_{0,0}$ is a special case. If the system is initially in state Π_0 (no processors blocked), it will remain in this state if either zero requests or one new request are issued. If zero new requests are issued, the bus will be idle for the cycle and the system will remain in state Π_0. If a single new request is issued, it will be serviced immediately by the bus and the system will also remain in state Π_0. Thus,

$$\pi_{0,0} = \alpha_{0,0} + \alpha_{0,1}$$

It was already explained that $\pi_{i,j} = 0$ if $i - j > 1$, since only one request can be served per cycle. For the case $i - j \leq 1$ (and i and j are not both zero), we have $\pi_{i,j} = \alpha_{i,1+j-i}$.

Given the transition probabilities, it is possible to solve for the state probabilities, π_i, although it does not seem possible to give a closed form representation of the general solution. To compute the state probabilities π_i, we define $w_i = \pi_i / \pi_0$. This results in a triangular system of linear equations. Thus, it is easier to solve for the w_i values than to solve for the π_i values directly. Furthermore, simplifying the problem to a triangular system reduces the complexity of the solution step from $O(n^3)$ to $O(n^2)$. It can be shown that

$$w_0 = 1$$

$$w_1 = \frac{1-(\alpha_{0,0}+\alpha_{0,1})}{\alpha_{1,0}} = \frac{1}{q^{N-1}} - (Np+q)$$

and, for $2 \leq i < N$,

$$w_i = \frac{w_{i-1} - \sum_{j=0}^{i-1} \alpha_{j,i-j} w_j}{\alpha_{i,0}} = \frac{w_{i-1}}{q^{N-i}} - \sum_{j=0}^{i-1} w_j \binom{N-j}{i-j} p^{i-j}$$

The state probabilities π_i must sum to one, and $\pi_i = w_i \pi_0$, so

$$\sum_{i=0}^{N-1} \pi_i = \sum_{i=0}^{N-1} w_i \pi_0 = \pi_0 \sum_{i=0}^{N-1} w_i = 1$$

therefore,

$$\pi_0 = \frac{1}{\sum_{i=0}^{N-1} w_i}$$

The mean queue length L (the mean number of blocked processors) is given by

$$L = \sum_{i=0}^{N-1} i\pi_i$$

The mean number of bus cycles needed to service a memory request, s, is one more than the mean queue length. The reason for this is that the queue length only represents the waiting time before a request is granted the bus; once the memory request has been granted the bus, it requires an additional cycle to service it. Therefore,

$$s = 1 + \sum_{i=0}^{N-1} i\pi_i \tag{3.2}$$

We define t_m to be the mean time interval between the memory requests from a processor. Each processor is assumed to continuously loop though the following sequence: compute, wait in queue for bus, use bus, access main memory. This mode of operation is shown in Figure 3.4. The average time for each pass through this loop is t_m. Since the goal of this chapter is to examine the bus performance, we divide the components of the mean time between memory references, t_m, into two groups: those components that are independent of the number of processors, N, and those components that are dependent on N. The compute time and the memory access time are independent of N. We represent the sum of these times by t_r. The bus queuing and bus transmission times, on the other hand, are dependent on N. We represent the sum of these times by t_b. The total time between memory accesses for a processor, t_m, is given by the sum of these components, so $t_m = t_b + t_r$.

• • •	Memory access time	Compute	Wait in queue for bus	Bus transmission time	Memory access time	Compute	• • •

Figure 3.4: Typical execution sequence for a processor

A more complex bus interaction can also be handled by this model. For example, consider the case in which each processor loops through the sequence shown in Figure 3.5, where both the processor and the memory are subject to bus queuing and bus transmission delays. In this case, t_r is still the sum of the compute time and the memory access time, and we let t_b be the sum of the bus queuing and transmission times for both the processor and the memory.

Since t_c is the bus cycle time, and a memory request takes s cycles to service, the mean time required for the bus to service a memory request, t_b, is given by, $t_b = st_c$.

• • •	Compute	Processor waits for bus	Processor uses bus	Memory access time	Memory waits for bus	Memory uses bus	• • •

Figure 3.5: More complex processor execution sequence

It follows from our earlier definition that the memory request probability p is equal to the ratio of the bus cycle time to the memory request time:

$$p = \frac{t_c}{t_m} = \frac{t_c}{t_b + t_r}$$

Substituting st_c for t_b yields

$$p = \frac{1}{s + \frac{t_r}{t_c}} \qquad (3.3)$$

Let v be given by

$$v = \frac{t_r}{t_c} \qquad (3.4)$$

Since t_r is the mean compute time between memory references, and t_c is the bus cycle time, v is a measure of the mean compute time in bus cycles. For each memory reference issued by the processor, v bus cycles of compute time and s bus cycles of memory service time elapse. Thus, the processor issues one memory request every $s+v$ bus cycles. Substituting equation (3.4) in equation (3.3) gives

$$p = \frac{1}{s + v} \qquad (3.5)$$

The bus utilization is defined as the fraction of bus cycles that are used to service a memory request. This will be a number between zero and one. Since the bus is in use unless the system is in state zero and no processors generate new requests, the bus utilization U is given by,

$$U = 1 - \pi_0 \alpha_{0,0} = 1 - \pi_0 q^N \tag{3.6}$$

Tables 3.1 and 3.2 show the results of this model for selected values of N and p.

	Memory request probability p								
N	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
2	0.20	0.39	0.57	0.72	0.83	0.92	0.97	0.99	1.00
4	0.39	0.72	0.91	0.98	1.00	1.00	1.00	1.00	1.00
6	0.57	0.93	1.00	1.00	1.00	1.00	1.00	1.00	1.00
8	0.74	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00
10	0.87	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
12	0.95	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
14	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
16	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 3.1: Bus utilization as a function of N and p

From Table 3.1, it can be seen that as p increases, the number of processors N at which the bus saturates decreases (the bus is saturated when the bus utilization reaches 1, indicating there are no idle bus cycles). From Table 3.2, it can be seen that once bus saturation is reached, s, the mean number of cycles for bus service, increases rapidly as N, the number of processors, increases. From these results, it is apparent that an excessively large value of N causes s to become unacceptably large. After a certain point, the decrease in performance due to increased bus time will more than offset the increase in performance from adding another processor.

	Memory request probability p								
N	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90
2	1.01	1.05	1.11	1.21	1.33	1.47	1.62	1.76	1.89
4	1.08	1.40	1.96	2.54	3.00	3.33	3.57	3.75	3.89
6	1.25	2.37	3.68	4.50	5.00	5.33	5.57	5.75	5.89
8	1.61	4.04	5.67	6.50	7.00	7.33	7.57	7.75	7.89
10	2.29	6.00	7.67	8.50	9.00	9.33	9.57	9.75	9.89
12	3.45	8.00	9.67	10.50	11.00	11.33	11.57	11.75	11.89
14	5.10	10.00	11.67	12.50	13.00	13.33	13.57	13.75	13.89
16	7.01	12.00	13.67	14.50	15.00	15.33	15.57	15.75	15.89

Table 3.2: Mean cycles for bus service s as a function of N and p

Using the results of this bus interference analysis, a numerical solution was obtained for the maximum number of processors. These results are shown in Table 3.3, which gives the maximum value of p for which N processors may be used in a system that is not loaded beyond the peak in its performance curve. In other words, the table shows the value of p below which a system with N processors will have a higher throughput than a system with $N - 1$ processors. Consider a system constructed with a value of p that exceeds the value given in the table for a particular value of N. The performance of such a system could actually be increased by decreasing the number of processors.

The system throughput T was defined previously as the ratio of the memory traffic for the multiprocessor system to the memory traffic for a single processor with a zero delay bus. This can be calculated from the bus cycle time t_c, the processor and memory time t_r, and the bus utilization U. The memory traffic for a single processor with $t_b = 0$ is $1/t_r$, and the memory traffic for the multiprocessor system is U/t_c, thus $T = Ut_r/t_c$.

N	Maximum p
2	0.999
3	0.454
4	0.268
5	0.188
6	0.144
7	0.117
8	0.098
9	0.084
10	0.074
11	0.066
12	0.059
13	0.054
14	0.049
15	0.046
16	0.042

Table 3.3: Maximum value of p for N processors

Substituting from (3.4) and (3.6) gives,

$$T = \left(1 - \pi_0 q^N\right) v \qquad (3.7)$$

Solutions of the simultaneous equations (3.2) and (3.5) may be obtained for given values of N and v. Equation (3.7) may then be used to obtain T as a function of N and v. Figure 3.6 illustrates the iterative solution process.

This process was found to converge quickly for realistic parameter values. For all of the examples considered in this chapter, the iterative solution process shown in Figure 3.6 can be completed in several hundred milliseconds on a Sun 4/260 processor.

3.4 Maximum throughput for a linear bus

In this section, we consider a bus in which the linear component of the bus delay, given by k_{lin}, is large compared with the other components of bus delay. This is the type of bus most likely to be encountered in a practical multiprocessor system; therefore, this bus will be considered in greater detail for the remainder of this chapter.

We assume that the bus has $N + 1$ connections, N processors and a single memory controller. The bus cycle time t_c for this bus is $k_{\text{lin}}(N + 1)$. To obtain an expression for p, we define the ratio:

$$r_{\text{lin}} = \frac{k_{\text{lin}}}{t_r}$$

Note that r_{lin} is dependent on both the bus technology and on the particular workload, but, since both k_{lin} and t_r are independent of N, r_{lin} is independent of N. Substituting $t_c = k_{\text{lin}}(N + 1) = t_r r_{\text{lin}}(N + 1)$ into (3.4), (3.5), and (3.7) gives

$$v = \frac{1}{r_{\text{lin}}(N + 1)}$$

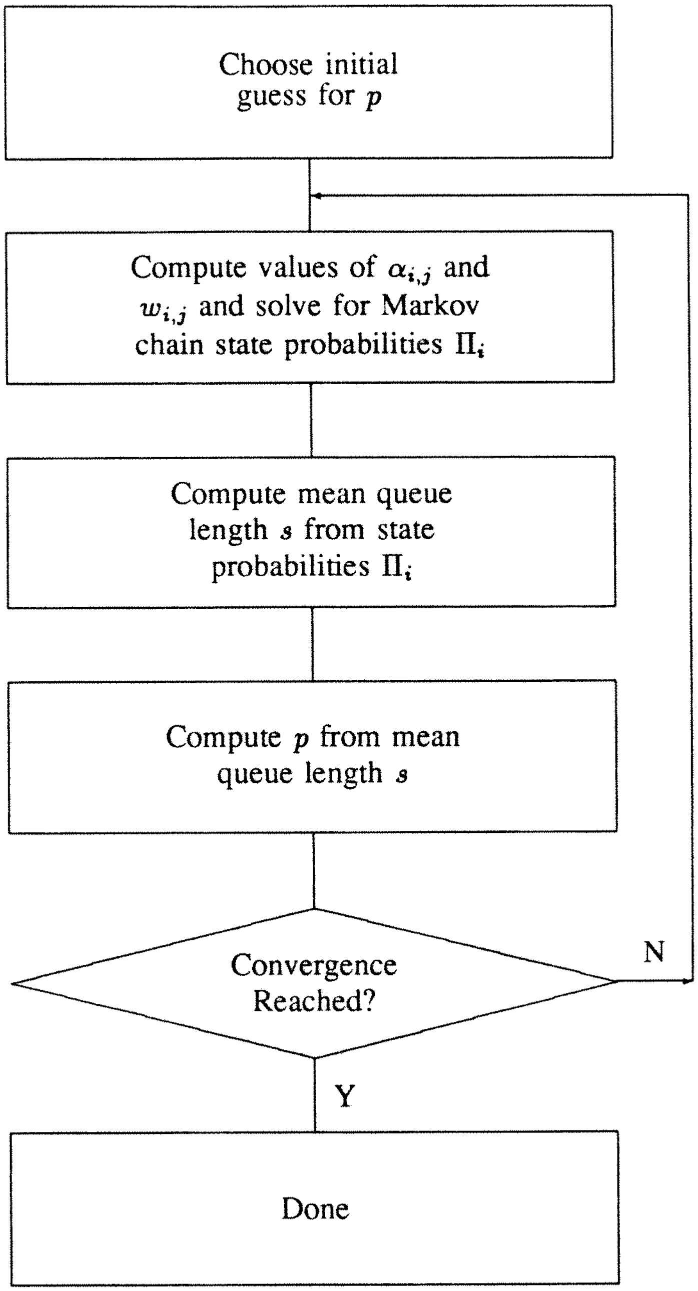

Choose initial guess for p
Compute values of $\alpha_{i,j}$ and $w_{i,j}$ and solve for Markov chain state probabilities Π_i
Compute mean queue length s from state probabilities Π_i
Compute p from mean queue length s
Convergence Reached?
N
Y
Done

Figure 3.6: Iterative solution for state probabilities

$$p = \frac{1}{s + \frac{1}{r_{\text{lin}}(N+1)}} \tag{3.8}$$

and,

$$T = \frac{1 - \pi_0 q^N}{r_{\text{lin}}(N+1)} \tag{3.9}$$

By solving the simultaneous equations (3.2), (3.8), and (3.9), the throughput T may be obtained as a function of the number of processors N and the ratio r_{lin}.

For a given value of r_{lin}, if the total system throughput is plotted as a function of the number of processors, it will be found to increase up to a certain number of processors and then decrease again when the bus becomes overloaded. This occurs because once the bus utilization U gets close to one, further increases in the number of processors N will not significantly increase U, but the bus cycle time, t_c, will increase due to increased bus loading. Thus, the throughput $T = Ut_r/t_c$ will decrease.

The effects of bus loading are illustrated in Table 3.4 where throughput is shown as a function of N for a particular r_{lin}. The throughput is plotted in Figure 3.7. It can be seen that as N increases, the mean time to service a memory request, s, also increases steadily. The throughput, T, on the other hand, increases as N increases until $N = 10$ and then it decreases again. The decrease is caused by the increased bus cycle time that results from the increased bus loading. The request probability, p, also increases with increasing N up to a point, but then decreases as the bus becomes slower.

There are two factors limiting the maximum throughput: the maximum total computation rate of the processors and the bus bandwidth. The maximum total computation rate increases linearly with the number of processors. The bus bandwidth, on the other hand, is inversely proportional to the number of processors, since it is

N	T	p	s
1	0.98	0.0196	1.00
2	1.94	0.0291	1.00
3	2.88	0.0385	1.00
4	3.79	0.0476	1.02
5	4.67	0.0565	1.04
6	5.49	0.0650	1.09
7	6.23	0.0731	1.18
8	6.84	0.0803	1.34
9	7.25	0.0863	1.59
10	7.42	0.0904	1.97
11	7.37	0.0927	2.46
12	7.16	0.0933	3.03
13	6.85	0.0927	3.65
14	6.51	0.0912	4.30
15	6.16	0.0893	4.95
16	5.84	0.0871	5.60
17	5.53	0.0847	6.25
18	5.25	0.0823	6.88
19	4.99	0.0799	7.51
20	4.76	0.0776	8.12

Table 3.4: T, p, and s as a function of N ($r_{\text{lin}} = 0.01$)

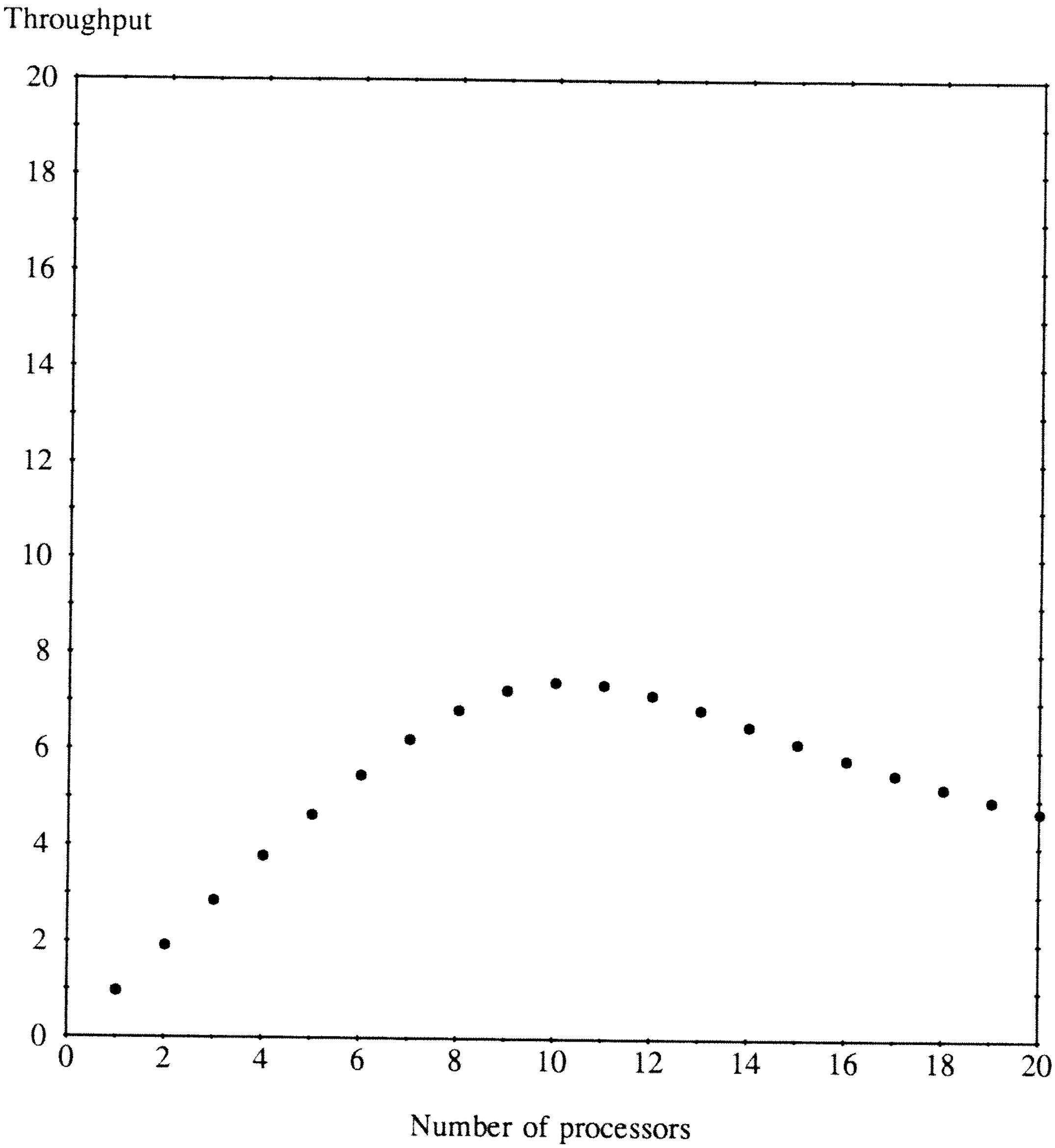


Figure 3.7: Throughput as a function of the number of processors

inversely proportional to the load on the bus, and the load on the bus was assumed to be directly proportional to the number of processors. Thus, the bus bandwidth decreases as the number of processors increases. Figure 3.8 shows these two limiting factors, again using a value of 0.01 for r_{lin}.

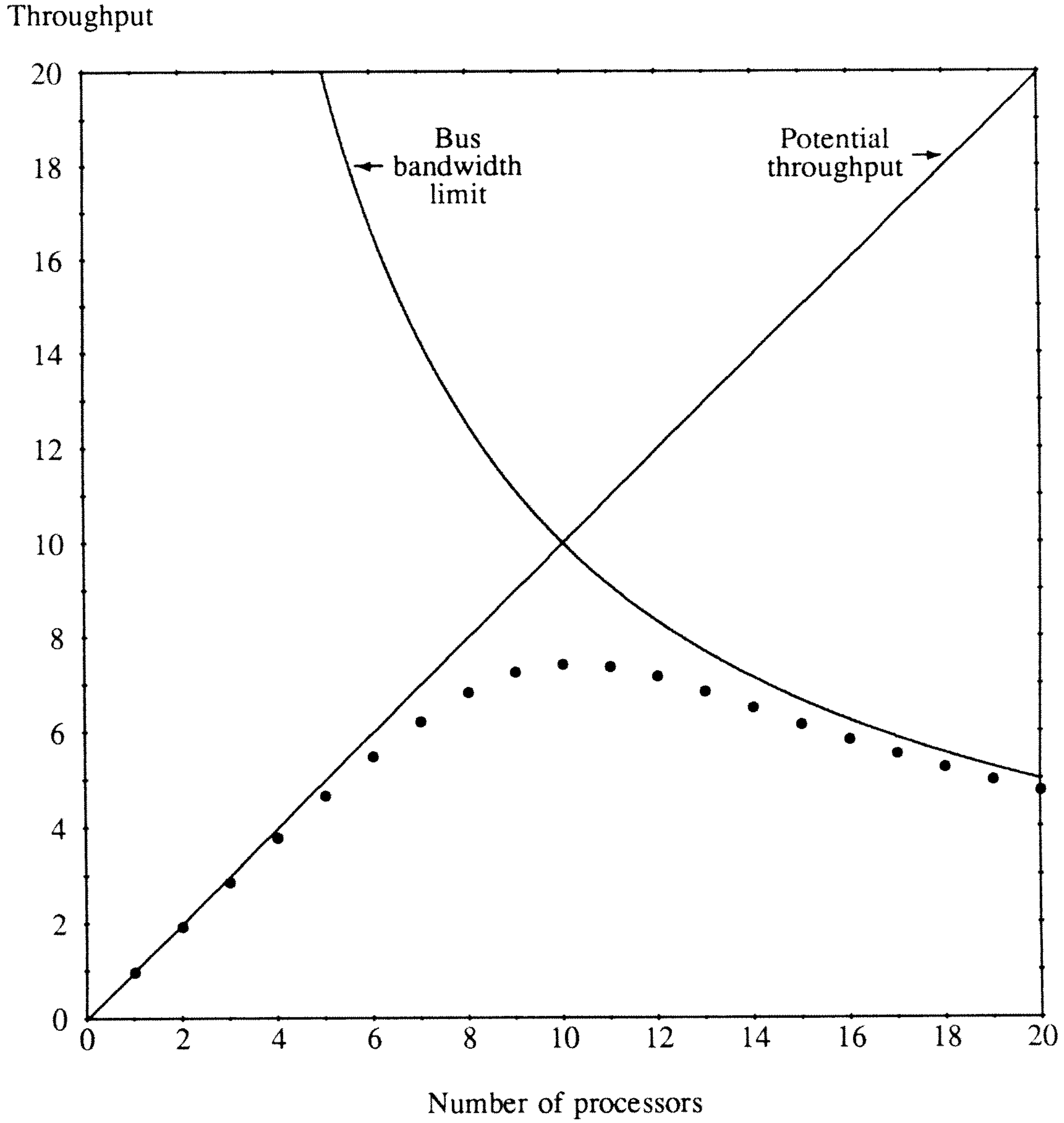


Figure 3.8: Asymptotic throughput limits

Using the results of the bus interference model described in the previous section, a solution was obtained for the maximum number of useful processors $N_{\max}$ as a function of r_{lin}. We define $N_{\max}$ to be the value of N at which the throughput of a system with $N+1$ processors is equal to that of a system with N processors for a particular value of r_{lin}. This is the value of N for which the maximum throughput is obtained. Table 3.5 shows $N_{\max}$ for a range of values of r_{lin}. The throughput, T, the request probability, p, and the mean number of bus cycles for service, s, are also shown. From Table 3.5, it can be shown that for small r_{lin}, the following approximation holds

r_{lin}	$N_{\max}$	T	p	s
0.192	2	1.11	0.346	1.15
0.0536	4	2.64	0.196	1.38
0.0146	8	5.92	0.107	1.73
0.00384	16	12.82	0.0569	2.28
0.00305	18	14.58	0.0509	2.39
0.000985	32	27.18	0.0295	3.09
0.000249	64	56.79	0.0151	4.28
0.000197	72	64.29	0.0135	4.53
0.0000622	128	117.35	0.00766	6.00
0.0000155	256	240.44	0.00386	8.45
0.0000123	288	271.43	0.00343	8.96
0.00000387	512	489.47	0.00194	11.94
0.000000964	1024	991.58	0.000972	16.88
0.000000761	1152	1117.53	0.000864	17.90

Table 3.5: $N_{\max}$ as a function of r_{lin} for a linear bus

$$r_{\text{lin}} \approx N_{\max}^{-2}$$

Since r_{lin} is specified by the processor, cache, and bus characteristics, this relationship gives an approximate idea of the largest useful system that can be constructed with these components. In other words, for a given value of $r_{\text{lin}} = \left(\frac{k_{\text{lin}}}{t_r}\right)$, the greatest number of processors that can be usefully incorporated into a system is given by:

$$N_{\text{max}} \approx \sqrt{\frac{1}{r_{\text{lin}}}}$$

or,

$$N_{\text{max}} \approx \sqrt{\frac{t_r}{k_{\text{lin}}}}$$

From this result, it can be seen that to double the maximum useful number of processors in a system with a single linear bus, it is necessary to increase t_r or decrease k_{lin} by a factor of four. For a given processor, t_r is approximately inversely proportional to the cache miss ratio. Thus, a decrease in cache miss ratio by a factor of four will approximately double N_{max}. The other factor, k_{lin}, is dependent on the technology used to implement the bus. Improving the bus speed by a factor of four will also approximately double N_{max}.

3.5 TTL bus example

In this section, we will illustrate the linear bus case with a system using standard TTL components for its bus transceivers. First, we show that a TTL bus is, in fact, a linear bus.

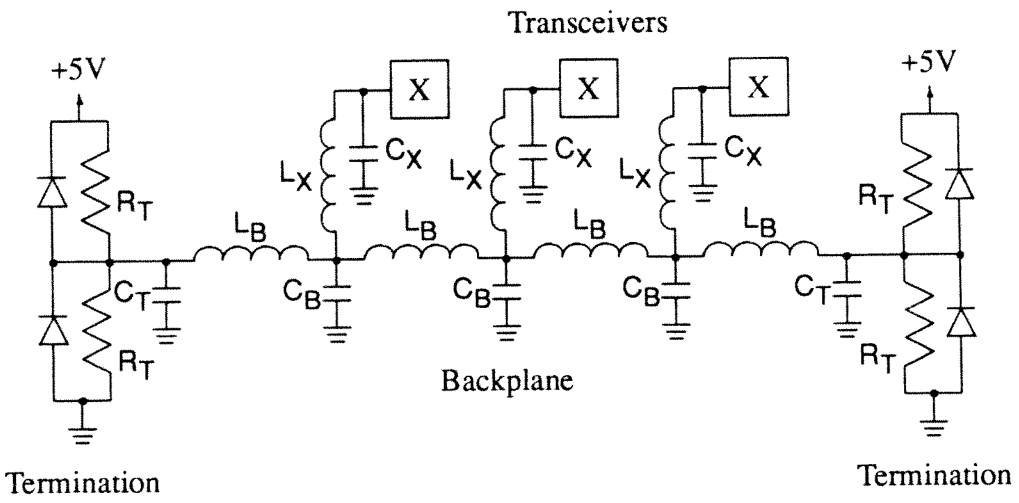
The delay of a TTL bus consists primarily of the time required for the driver to charge the transceiver capacitances. The total capacitance which must be charged by the driver is proportional to the number of transceivers connected to the bus. Since the driver acts as a nearly constant current source, the delay is nearly proportional to the number of transceivers connected to the bus. Thus, the k_{lin} term will be dominant in the bus delay model.

An estimate of the bus delay for a large bus may be made as follows. We assume a 64 ma FAST Schottky TTL bus transceiver. Typical characteristics for this component are 12 pf output capacitance, 64 ma driver current, and 2 volt logic swing [Motor83]. Assuming 1 pf per connection of additional capacitance from the bus wiring gives a total of 13 pf per connection. With terminations of 64 ohms to 2.5 volts, and a logic low level of 0.6 volts, approximately 30 ma of driver current flows through the terminations, leaving 34 ma to charge the capacitances. From these figures, we get

$$k_{\text{lin}} = \frac{\Delta}{N} \approx \frac{(13\,\text{pf})\,(2\,\text{V})}{(34\,\text{ma})} = 0.76\,\text{ns}$$

To obtain a more precise estimate of the delay of a TTL bus, a circuit model was constructed in which the driving transceiver was modeled as a nonlinear current source in parallel with a capacitor, and the receiving transceivers were modeled as nonlinear resistors in parallel with capacitors. The bus was modeled with lumped inductances and capacitances. The distributed inductance for each interconnection wire was represented by a single inductor, and the distributed capacitance for each wire was represented by placing half of the total distributed capacitance at each end of the wire. Using this model, the bus was represented by a system of nonlinear ordinary differential equations. Parameters for this model were obtained from [Motor83]. Figure 3.9 illustrates the circuit model used.

The ODEPACK (Ordinary Differential Equation Package) software package was used to obtain a solution to this system, from which the bus delay was determined.



L_B = inductance of backplane wiring per bus slot

L_X = inductance of stub to transceiver

C_B = capacitance of backplane wiring per bus slot plus $\frac{1}{2}$ capacitance of stub to transceiver

C_X = capacitance of transceiver plus $\frac{1}{2}$ capacitance of stub to transceiver

C_T = $\frac{1}{2}$ capacitance of backplane wiring per bus slot

R_T = 2 times bus characteristic impedance

Figure 3.9: Bus circuit model (N = 3)

The results for buses with 25 to 96 connections on them are shown in Table 3.6. For smaller N, effects due to transmission line reflections dominate, and the delay model tends to be somewhat unpredictable. For larger N, the computation time required for the simulations becomes prohibitive.

N	Delay (ns)
25	26.1
26	27.3
27	28.4
28	29.6
29	30.7
30	31.2
31	32.3
32	33.5
34	35.7
36	37.4
38	39.7
40	41.9
42	43.6
44	45.9
46	48.1
48	49.8
52	54.3
56	58.3
60	62.3
64	66.2
68	70.7
72	74.6
76	78.6
80	83.1
84	87.1
88	91.0
92	94.9
96	99.4

Table 3.6: Bus delay as calculated from ODEPACK simulations

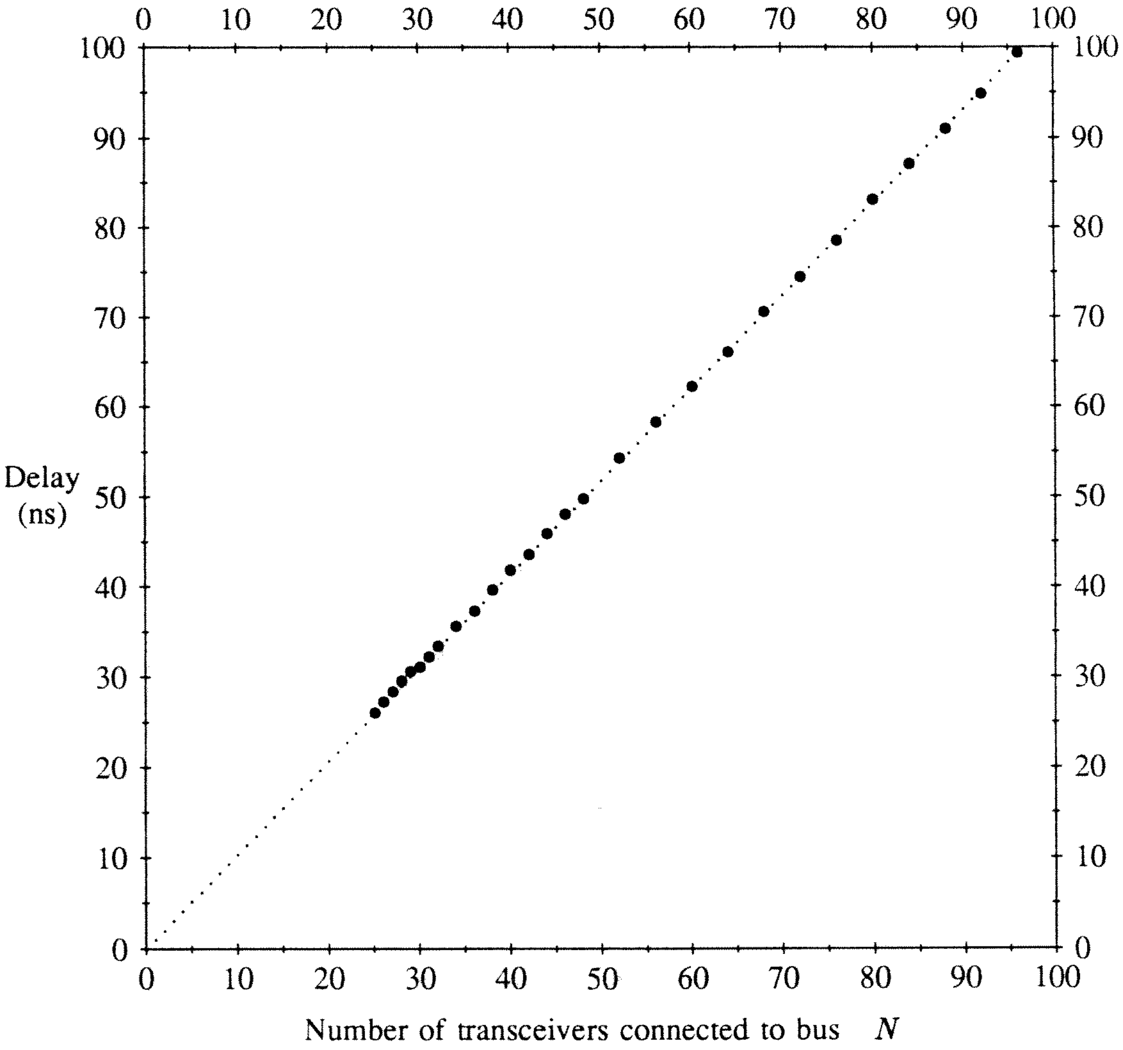


Figure 3.10: Bus delay as calculated from ODEPACK simulations

The results given in Table 3.6 are plotted in Figure 3.10. From the figure, it can be seen that for large N ($N \geq 25$), the delay is essentially linear in the number of bus connections, with $k_{\text{lin}} \approx 1.04$ ns. Thus, a linear bus model may be used for a TTL bus. The value for k_{lin} from Figure 3.10 is slightly larger than the value estimated from the transceiver capacitance and drive current, and it represents the typical time to reach correct logic values everywhere on the bus. A practical implementation must use a time significantly greater than this figure to allow for worst case variations, clock skew, arbitration time, etc. To account for these in our examples, we consider existing TTL bus designs. Several TTL based buses are described in [Borri85]. The maximum bandwidth claimed for any of these buses is 57 megabytes per second with 21 bus slots. This bus is four bytes wide, so we have

$$k_{\text{lin}} = \frac{(4 \text{ bytes})}{(57 \times 10^6 \text{ bytes/sec})(21)} = 3.34\,\text{ns} \qquad (3.10)$$

We will use this figure for k_{lin} for our examples that follow, and we will assume a linear model as Figure 3.10 would suggest.

To illustrate the use of this bus model in evaluating a practical system design, consider a hypothetical system with the following characteristics:

- 16.67 MHz Motorola MC68020 processor
- 64 Kbyte cache
- 8 byte bus data path
- 16 byte cache line size
- 3 bus cycles needed to fetch data for a cache miss

- 3 bus cycles needed to write a dirty line back from the cache
- 260 ns main memory access time
- 7.0 ns bus transceiver propagation delay

The following 16.67 MHz 68020 performance figures are from [MR85]:

- 2.52 million instructions per second
- 1.201 memory references per instruction
- 52.1% of memory references are data references

The following cache performance figures are from [Smith85a] and [Smith87b]:

- Cache miss ratio = 0.030
- Half of data misses require writing a dirty line back from the cache

With this data, an estimate may be obtained for t_r:

$$t_r = \frac{\frac{1}{(2.52\times 10^6/\mathrm{s})(1.201)(0.030)} + 2.60 \times 10^{-7}\,\mathrm{s} + (2)(7.0 \times 10^{-9}\,\mathrm{s})}{3 \;+\; (3)(0.521)(0.5)} = 2.98\,\mu\mathrm{s}$$

The first term in the numerator gives the mean processor compute time between references (this is the reciprocal of instructions per second × references per instruction × miss ratio), the second term is the fixed memory access time, and the last term is the fixed round trip bus transceiver delay. The denominator is the mean number of bus cycles for a single memory reference and is the sum of the number of bus cycles needed to service a cache miss (three) and the average number of cycles to write back dirty lines. Three bus cycles are needed to fetch data or write a dirty line because one is required to issue the address and two are needed to transfer the data. From the figure obtained earlier for k_{lin} in (3.10), we get $r_{\mathrm{lin}} = k_{\mathrm{lin}}/t_r = 0.00112$. With this value of r_{lin}, a maximum throughput of 25.4 can be achieved using 30 processors.

3.6 Optimization of a two-level bus hierarchy

We now consider a bus that is implemented with two physical levels, as in the design of Figure 3.1. Using a delay model as described previously to model each level, it is possible to select the number of interconnections that should be made at each level in order to minimize the total delay. For example, to build a 256 processor system, the following are plausible approaches: give each processor its own interface to a 256 slot bus, connect pairs of processors to a 128 slot bus, or connect groups of 16 processors to a 16 slot bus. In this section, a method will be developed for selecting the organization with the minimum delay.

If we let N be the total number of devices connected through both levels and B be the number of devices connected together at the first level, then N/B devices must be connected together at the second level. The optimization problem is to choose B to minimize the total delay, Δ, for a given N. The longest path in this hierarchy is from a processor through its level one bus, down through the level two bus, and back up to a different level one bus (see Figure 3.1). It is necessary to include this last delay to allow the processors on that bus to perform their snooping operations. Thus, the delay of the two-level hierarchy is twice the level one delay plus the level two delay, i.e.,

$$\Delta = 2\Delta_1 + \Delta_2$$

where

$$\Delta_1 \;=\; k_{\text{const}_1} \;+\; k_{\log_1} \log_2 B \;+\; k_{\text{lin}_1} B \;+\; k_{\text{quad}_1} B^2$$

and

$$\Delta_2 \;=\; k_{\text{const}_2} \;+\; k_{\log_2} \log_2 \left(\frac{N}{B}\right) \;+\; k_{\text{lin}_2} \left(\frac{N}{B}\right) \;+\; k_{\text{quad}_2} \left(\frac{N}{B}\right)^2$$

Therefore,

$$\begin{aligned}\Delta \;=\;& 2k_{\text{const}_1} \;+\; 2k_{\log_1} \log_2 B \;+\; 2k_{\text{lin}_1} B \;+\; 2k_{\text{quad}_1} B^2 \;+\\ & k_{\text{const}_2} \;+\; k_{\log_2} \log_2 \left(\tfrac{N}{B}\right) \;+\; k_{\text{lin}_2} \left(\tfrac{N}{B}\right) \;+\; k_{\text{quad}_2} \left(\tfrac{N}{B}\right)^2\end{aligned}$$

Taking the derivative of Δ with respect to B and setting the result to zero gives

$$4k_{\text{quad}_1} B^4 \;+\; 2k_{\text{lin}_1} B^3 \;+\; (2k_{\log_1} - k_{\log_2}) B^2 \;-\; k_{\text{lin}_2} N B \;-\; 2k_{\text{quad}_2} N^2 \;=\; 0$$

Although a closed form solution of this quartic equation is possible, it is complex and gives little additional insight into the problem. However, it is useful to consider the cases in which a particular type of delay is dominant. In these cases, simpler approximations are possible. The value of B that will minimize the total delay for all combinations of level one and level two delays is shown in Table 3.7. The situation in which the dominant delays in level one and level two are different may occur when the levels are implemented in different technologies. For example, one level may be implemented entirely within a single VLSI chip, while another level may consist of connections between the chips.

3.7 Maximum throughput for a two-level bus hierarchy

Using Table 3.7, we can determine the number of processors for maximum throughput when the two-level bus hierarchy of Figure 3.1 is used instead of a single bus. We will

	Level one constant	Level one logarithmic	Level one linear	Level one quadratic
Level two constant	Constant delay; B doesn't matter	1	1	1
Level two logarithmic	N	1 if $2k_{\log_1} > k_{\log_2}$ N if $2k_{\log_1} < k_{\log_2}$	$\frac{k_{\log_2}}{2k_{\mathrm{lin}_1}}$	$\sqrt{\frac{k_{\log_2}}{4k_{\mathrm{quad}_1}}}$
Level two linear	N	$N\frac{k_{\mathrm{lin}_2}}{2k_{\log_1}}$	$\sqrt{N\frac{k_{\mathrm{lin}_2}}{2k_{\mathrm{lin}_1}}}$	$\sqrt[3]{N\frac{k_{\mathrm{lin}_2}}{4k_{\mathrm{quad}_1}}}$
Level two quadratic	N	$N\sqrt{\frac{k_{\mathrm{quad}_2}}{k_{\log_1}}}$	$\sqrt[3]{N^2\frac{k_{\mathrm{quad}_2}}{k_{\mathrm{lin}_1}}}$	$\sqrt{N\sqrt{\frac{k_{\mathrm{quad}_2}}{2k_{\mathrm{quad}_1}}}}$

Table 3.7: Value of B for minimum delay in a two-level bus hierarchy

assume the electrical characteristics of the two levels are identical and linear. Then, the bus cycle time at each level is approximately k_{lin} times the number of transceivers connected at that level. Furthermore, the optimal organization for N processors is to connect them as $\sqrt{2N}$ clusters with $\sqrt{N/2}$ processors in each cluster. The level one buses will have $\sqrt{N/2}$ processor connections and one connection to the level two bus for a total of $\sqrt{N/2}+1$ connections. Similarly, the level two bus will have $\sqrt{2N}$ connections to the level one buses and one connection to the main memory for a total of $\sqrt{2N}+1$ connections. The bus cycle time is twice the level one delay plus the level two delay, so $t_c = k_{\mathrm{lin}}(\sqrt{8N}+3)$. Substituting $t_c = k_{\mathrm{lin}}(\sqrt{8N}+3) = t_r r_{\mathrm{lin}}(\sqrt{8N}+3)$ into (3.4), (3.5), and (3.7) gives

$$v = \frac{1}{r_{\mathrm{lin}}\left(\sqrt{8N}+3\right)}$$

$$p = \frac{1}{s + \frac{1}{r_{\mathrm{lin}}(\sqrt{8N}+3)}} \tag{3.11}$$

$$T = \frac{1-\pi_0 q^N}{r_{\mathrm{lin}}\left(\sqrt{8N}+3\right)} \tag{3.12}$$

By solving the simultaneous equations (3.2), (3.11), and (3.12), the throughput, T, may be obtained as a function of the number of processors N and the ratio r_{lin}. As was done for a single level bus, the value of N_{max} was determined for a range of values of r_{lin}. Table 3.8 shows these results, along with the throughput, T, the request probability, p, and the mean number of bus cycles for service, s.

It can be shown that for small r_{lin}, the following approximation holds

$$r_{\mathrm{lin}} \approx (2N_{\mathrm{max}})^{-\frac{3}{2}}$$

r_{lin}	$N_{\max}$	T	p	s
0.0130	8	5.66	0.113	1.85
0.00418	16	13.97	0.0537	2.68
0.00182	32	26.32	0.0308	3.50
0.000551	72	63.05	0.0138	5.13
0.000235	128	115.79	0.00780	6.76
0.0000705	288	269.28	0.00347	10.02
0.0000299	512	486.78	0.00195	13.27
0.00000893	1152	1113.78	0.000868	19.77

Table 3.8: $N_{\max}$ as a function of r_{lin} for two levels of linear buses

Thus,

$$N_{\max} \approx \frac{1}{2} r_{\text{lin}}^{-\frac{2}{3}}$$

From this result, it can be seen that to double the maximum useful number of processors in a system with a two-level hierarchy of linear buses, it is necessary to increase t_r or decrease k_{lin} by a factor of about 2.83 ($2^{\frac{3}{2}}$). This is significantly better than the single linear bus case considered previously where a factor of four improvement in r_{lin} is needed to double $N_{\max}$. As before, this relation gives a simple approximation for the largest useful system that may be constructed with a given processor, cache, and two-level bus.

For moderate to large N, much better throughput can be obtained with a two-level hierarchy than with a single level. In the previous single level linear bus example, with $r_{\text{lin}} = 0.00112$, it was found that a maximum throughput of 25.4 could be achieved using 30 processors. If a two-level bus hierarchy is constructed using the same technology (and thus the same value of r_{lin}), then a maximum throughput of 37.8 can be achieved using 50 processors. This represents a 49% improvement in maximum throughput over the single level bus. When the number of processors N is small, however, the improvement is not significant, and for $N < 12$, throughput is actually worse for a two-level bus hierarchy than for a single level bus.

3.8 Maximum throughput using a binary tree interconnection

Finally, in this section, we determine the number of processors for maximum throughput when the binary tree interconnection network shown in Figure 3.2 is used. In this case, the logarithmic component of the bus delay, given by $k_{\log}$, is large compared with the other components of bus delay. For this system, $t_c = k_{\log} \log_2 N$. Defining $r_{\log} = k_{\log}/t_r$ gives

$$t_c = t_r r_{\log} \log_2 N$$

$$v = \frac{1}{r_{\log} \log_2 N}$$

$$p = \frac{1}{s + \frac{1}{r_{\log} \log_2 N}} \qquad (3.13)$$

$$T = \frac{1 - \pi_0 q^N}{r_{\log} \log_2 N} \qquad (3.14)$$

The solution for this case is shown in Table 3.9, and the approximation for small $r_{\log}$ is

$$r_{\log} \approx \frac{1}{N_{\max} \log_2 N_{\max}}$$

As N becomes large, the maximum number of processors can be doubled by decreasing r_{lin} by a factor of slightly more than 2. This is better than either of the two previous bus to give worse performance than the two-level bus hierarchy for practical numbers of processors.

3.9.1 Single bus example

We assume the same processor, cache, memory, and workload characteristics as in the previous TTL bus examples, thus we still have $t_r = 2.98\,\mu\text{s}$. This gives $r_{\text{lin}} = k_{\text{lin}}/t_r = 0.000228$. Using a single bus with this value of r_{lin}, a maximum throughput of 59.5 can be achieved using 67 processors. In the example with a single-level TTL bus, it was found that a maximum throughput of 25.4 could be achieved using 30 processors. Thus, an improvement in maximum throughput by a factor of 2.3 has been obtained simply by using a bus with better electrical characteristics.

3.9.2 Two-level bus example

Again using the figure $r_{\text{lin}} = 0.000228$, but applying the two-level bus model this time, we find that a maximum throughput of 118.8 can be achieved using 136 processors (the optimal configuration has 17 clusters with 8 processors in each cluster). In the example with a two-level TTL bus, it was found that a maximum throughput of 37.8 could be achieved using 50 processors. In this case, an improvement in maximum throughput by a factor of 3.1 has been obtained by improving the electrical characteristics of the bus.

From these examples based on the Futurebus technology, it is apparent that the bus electrical characteristics have a major impact on the maximum performance of these bus-oriented shared memory multiprocessor systems.

3.10 Summary

In this chapter, we have introduced the concept of a logical bus, an interconnection network that preserves the logical structure of a single bus while avoiding the electrical implementation problems associated with physically attaching all of the processors directly to a single bus. We have described two examples of logical buses, a two-level bus hierarchy and a binary tree structure.

We have developed a new Markov chain model for multiprocessor performance that accurately considers the effects of bus loading of system performance. Using this model, we have derived estimates for the maximum useful number of processors as a function of the processor and bus characteristics in systems using single-level bus, two-level bus, and binary tree interconnections.

Using one example based on TTL bus transceivers such as those used in the standard VME bus, and another example based on transceivers designed for the IEEE P896 Futurebus, we have demonstrated the importance of good bus electrical design.

CHAPTER 4

TRACE DRIVEN SIMULATIONS

To validate the performance model developed in the previous chapter, simulations based on address traces were used. This chapter describes the simulation models used, the workloads for which traces were obtained, and the results of these simulations.

Two types of workload were used. The first workload represented a multiprogramming environment in which each processor works on an independent task. This model of execution is typical of what might be observed in an environment in which many different jobs or many instances of the same job are independently running at the same time. Time slicing context switches are infrequent in this environment, since a relative long time quantum can be used in a multiprocessor without degrading interactive response time unacceptably, and most processes will run to completion or block before the time quantum is exceeded.

The second workload represented a parallel algorithm in which all processors cooperated in solving a single problem. The algorithm chosen was an example of SCMD (single code, multiple data) parallelism. Each processor is loaded with the same program but works on a different portion of the data, and the processors do not run in lockstep.

4.1 Necessity of simulation techniques

When using a probabilistic model to approximate the behavior of a computer running a real workload, it is necessary to verify that the model is truly representative of the workload. For estimating cache memory performance, no probabilistic model is generally acceptable. It has been observed that: "there do not currently exist any generally accepted or believable models for those characteristics of program behavior that determine cache performance" [Smith85a]. Therefore, trace driven simulations are the usual method for evaluating the performance of proposed cache designs. A detailed discussion of trace driven simulation techniques for caches is presented in [Smith85a].

Based on this experience with cache memories, similar concerns could be raised over the probabilistic bus models presented in Chapter 3, particularly in light of the independence assumption. To validate the models used in Chapter 3, two simulation systems were implemented so that the performance of actual parallel workloads could be compared with the performance predicted by the models. The first simulation system was designed for a multiprogramming workload in which each processor works on an independent task. The assumption that the memory references of different processors are independent is the key assumption used in the design of the Markov chain model. With a multiprogramming workload, this assumption should hold, since each processor is running a separate task. Thus, we expect the Markov chain model to accurately reflect the performance of the system when running a multiprogramming workload.

To investigate the effects of violating the independence assumption, a second simulator was designed to simulate a parallel algorithm in which all processors cooperated on the same problem. In this situation, memory references by different processors are not independent, so the assumptions underlying the Markov chain model do not hold. The results from this simulation are useful in determining how robust the Markov chain model is when its underlying assumption of independence is violated.

4.2 Simulator implementation

To obtain the simulation data, two different software tracing methods were used. The first trace generator was written to single step the Motorola MC68020 processor in a Sun 3/280 system. The traces obtained with this system were used to simulate a multiprogramming environment in which each processor ran an independent task. The second simulation used a complete software simulation of a hypothetical multiprocessor system based on the recently announced Motorola MC88100 microprocessor. This second simulator implementation was used to simulate the execution of a parallel algorithm in which all processors cooperate on a single problem. Each of these trace generation schemes is described in more detail in the following sections.

4.2.1 68020 trace generation and simulation

The first simulation implemented consisted of two components, a trace generator and a trace driven multiprocessor simulator. The trace generator was used to obtain an address trace for the selected workload. This address trace was then used as input for the multiprocessor simulator, which determined the expected throughput for a multiprocessor running the chosen workload.

The trace generator ran on a Sun 3/280 system which has a 25 MHz Motorola MC68020 processor. It used the Unix **ptrace** facility to step the traced program one instruction at a time. At each instruction, the trace generator performed the following steps:

1. Read the program counter to determine the address of the current instruction.

2. Fetch this instruction.

3. Decode the instruction and fetch additional instruction words if it is a multiword instruction.

4. Interpret all memory references made by the instruction.

Since the MC68020 processor has several complex addressing modes, interpreting the memory references made by an instruction can involve considerable effort. For example, obtaining one operand of an instruction could require obtaining a base address from an address register, adding a displacement from the instruction stream, adding a scaled index value from another register, fetching an indirect address from memory, adding another displacement and fetching the actual operand.

The trace generation program consists of approximately 3000 lines of C code. About 400 lines of this is machine independent code used to set up the program to be traced and to implement the single step mechanism. The remainder is machine dependent routines used to decode the complex instruction formats and addressing modes of the MC68020. From these figures, it is apparent that the complexity of a software address trace generation program depends strongly on the complexity of the architecture it is simulating.

The principal limitation of this trace generation method is that it only obtains the sequence of memory references; it cannot obtain timing information to indicate when each reference occurs. Thus, it is necessary to make an assumption about the timing of the references. The mean interval between references for the MC68020 processor has been found experimentally to be approximately one reference every six CPU clocks [MR85]. This figure of six clocks was used as a mean time between references for the MC68020 simulation experiments. No other data was available on the distribution of the time between references. The MC68020 hardware implementation imposes a minimum of three clocks between references. A simple processing module design might divide the processor clock rate by three to cycle the cache, thus constraining the time between cache references to be a multiple of three clocks. A more complex timing arrangement could allow any number of clocks to occur between cache references, subject to the three clock minimum of the MC68020.

To investigate the effect of the distribution of the times on simulated throughput, seven different distribution functions for the number of clocks between references were considered. All of the functions were selected to have a mean of six and a minimum of at least three. The selected distribution functions are shown in Table 4.1. The first distribution function, "deterministic", assumed references always took six clocks. The second function, "uniform synchronous", assumed that the number of clocks was constrained to be a multiple of three and that the distribution was uniform. The third function, "uniform asynchronous", also assumed a uniform distribution, but removed the restriction that the number of clocks be a multiple of three. The fourth and fifth functions, "geometric synchronous" and "geometric asynchronous", used geometric distributions, with and without the multiple of 3 constraint, respectively. The sixth and seventh functions, "Poisson synchronous" and "Poisson asynchronous", used Poisson distributions, with and without the multiple of 3 constraint, respectively. All seven discrete probability distribution functions for the number of clocks between references are tabulated in Table 4.2.

Name	Description
Deterministic	The constant value 6
Uniform Synchronous	A random variable chosen uniformly from the set {3, 6, 9}
Uniform Asynchronous	A random variable chosen uniformly from the set {3, 4, 5, 6, 7, 8, 9}
Geometric Synchronous	3 plus 3 times a geometrically distributed random variable with mean 1
Geometric Asynchronous	3 plus a geometrically distributed random variable with mean 3
Poisson Synchronous	3 plus 3 times a Poisson distributed random variable with mean 1
Poisson Asynchronous	3 plus a Poisson distributed random variable with mean 3

Table 4.1: Experimental time distribution functions

Clocks	Determ.	Uniform Synch.	Uniform Asynch.	Geometric Synch.	Geometric Asynch.	Poisson Synch.	Poisson Asynch.
1	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2	0.000	0.000	0.000	0.000	0.000	0.000	0.000
3	0.000	0.333	0.143	0.500	0.250	0.368	0.050
4	0.000	0.000	0.143	0.000	0.188	0.000	0.149
5	0.000	0.000	0.143	0.000	0.141	0.000	0.224
6	1.000	0.333	0.143	0.250	0.105	0.368	0.224
7	0.000	0.000	0.143	0.000	0.079	0.000	0.168
8	0.000	0.000	0.143	0.000	0.059	0.000	0.101
9	0.000	0.333	0.143	0.125	0.044	0.184	0.050
10	0.000	0.000	0.000	0.000	0.033	0.000	0.022
11	0.000	0.000	0.000	0.000	0.025	0.000	0.008
12	0.000	0.000	0.000	0.062	0.019	0.061	0.003
13	0.000	0.000	0.000	0.000	0.014	0.000	0.001
14	0.000	0.000	0.000	0.000	0.011	0.000	0.000
15	0.000	0.000	0.000	0.031	0.008	0.015	0.000
16	0.000	0.000	0.000	0.000	0.006	0.000	0.000
17	0.000	0.000	0.000	0.000	0.004	0.000	0.000
18	0.000	0.000	0.000	0.016	0.003	0.003	0.000
19	0.000	0.000	0.000	0.000	0.003	0.000	0.000
20	0.000	0.000	0.000	0.000	0.002	0.000	0.000
21	0.000	0.000	0.000	0.008	0.001	0.001	0.000
22	0.000	0.000	0.000	0.000	0.001	0.000	0.000
23	0.000	0.000	0.000	0.000	0.001	0.000	0.000
24	0.000	0.000	0.000	0.004	0.001	0.000	0.000
25	0.000	0.000	0.000	0.000	0.000	0.000	0.000
26	0.000	0.000	0.000	0.000	0.000	0.000	0.000
27	0.000	0.000	0.000	0.002	0.000	0.000	0.000
28	0.000	0.000	0.000	0.000	0.000	0.000	0.000
29	0.000	0.000	0.000	0.000	0.000	0.000	0.000
30	0.000	0.000	0.000	0.001	0.000	0.000	0.000

Table 4.2: Experimental probability distribution functions

Simulations were run using all seven distribution functions. It was found that the differences in predicted throughput between these distributions was insignificant in all cases. The largest difference observed between two simulations of systems that were identical except for their distribution functions was 0.12%. This is completely negligible; some of the input parameters for the model are not even known to this degree of precision.

Another limitation of this trace generation method is that it cannot obtain parallel traces. Since traces are obtained from a single process running on a uniprocessor, only a single thread of execution can be traced at any one time. However, this does not preclude the use of the traces for simulation of a multiprocessor system operating in a multiprogramming mode. One method is to obtain several different uniprocessor traces and then simulate each trace on a different processor of the multiprocessor. A slight variation on this is to run the same trace on all processors, but with each processor running at a different location in the trace, independently of the other users.

A third limitation of this trace generation method is that it only traces user program memory references; it cannot trace memory references made by the operating system. When the operating system code that implements a system call executes, it will displace some lines in the cache. As a consequence of this, the miss ratio is likely to increase following the system call, because some of these displaced lines will need to be reloaded into the cache. It is not possible to exactly determine what the effect of system calls will be on the miss ratio, since this trace generation method does not record instructions executed within the operating system. However, system calls were found to be infrequent in the workload used; only 224 system calls were observed in 1,775,947 memory references. Thus, the inability to trace system calls is not considered to be too severe of a limitation for this trace generation scheme and workload.

The multiprocessor simulator used to obtain performance measures from the workload address traces was designed to run on the same Sun 3/280 system used to perform the trace generation. It consisted of approximately 800 lines of C++ code. It consisted of a set of modules implementing the functionality of each of the hardware resources in the system (processors, caches, buses, and memory modules), along with a queue manager used to control system timing and to manage multiple requests to shared resources.

The simulated execution of the workload proceeded as follows. The trace was treated as a single infinite loop. Each processor started at a different location in this loop. The starting locations were roughly equally spaced throughout the trace, although it was found that the choice of starting location made almost no difference in the results. The simulator ran all of the processors in parallel, each executing at its own particular location in the trace.

4.2.2 88100 trace generation

This, the second trace generation scheme, was based on a complete software simulation of a hypothetical multiprocessor system based on the Motorola MC88100 microprocessor. The simulator consists of approximately 4000 lines of C code. The results presented here were run on a Sun 4/200 series host, although it was written to allow it to operate on almost any 32 bit processor running some derivative of Berkeley

Unix. Since it is a simulation, the results obtained are completely independent of the particular machine the simulation is hosted on.

The principal limitation of this trace generation scheme is that no operating system is available for the simulated system. Therefore, programs to be traced must be set up so as to not require any operating system services.

4.3 Simulation workload

The workload selected for the simulation models consisted of seven programs as follows:

- **cpp**, the C preprocessor
- **ccom**, the C compiler
- **c2**, the C peephole optimizer
- **as**, the MC68020 assembler
- **ld**, the object file linker
- **ls**, directory listing utility
- **dgefa**, matrix LU factorization by Gaussian elimination

The first six are all standard programs supplied with the Unix operating system. They were chosen because they are among the most frequently executed commands in a typical MC68020 based Unix programming development environment, and they span a wide range of program size and complexity. All of these programs were traced using the MC68020 trace generation program. The seventh program, **dgefa**, is based on the

Fortran subroutine of the same name in the LINPACK linear algebra subroutine package. This program is a double precision version of a routine to solve a linear system using the Gaussian elimination method of LU factorization. The original Fortran program was rewritten in C and the algorithm was modified for parallel execution on a shared memory multiprocessor. Unlike the other test programs, this program represents a true parallel algorithm. A run of the parallelized **dgefa** algorithm on a 64 by 64 dense matrix with random elements was traced using the MC88100 multiprocessor simulator.

To allow the results of the Markov chain model to be compared with the simulation results under realistic conditions, two hypothetical multiprocessor systems were investigated, one based on the 68020 processor and one based on the 88100 processor. The results for each of these systems are discussed in the following sections.

4.4 Results for 68020 example system

The example system using the 68020 processor was designed based on the following assumptions:

- The system contains 1 to 64 processors.
- Each processor is a Motorola MC68020 running at 25 MHz.
- Each processor has a 64 Kbyte direct mapped cache.
- The bus data path width is 8 bytes.
- The cache line size is 16 bytes.
- 3 bus cycles are needed to fetch data for a cache miss.

- 3 bus cycles are needed to write a dirty line back from the cache.

- The main memory access time is 160 ns.

- The total bus transceiver delay is 14 ns.

- Each CPU makes a memory reference once every 6 clocks (240 ns) on average.

- The cache miss ratio is 0.01486 (average for all traces).

- 34.96% of cache misses require a write back of a dirty line.

- The bus delay is linear with the number of processors, with $k_{\text{lin}} = 3.34\text{ns}$.

These assumptions can be justified from practical design considerations. The range 1 to 64 processors spans the full range of practical single bus systems that may be constructed using the selected technology. With one processor, the hypothetical system is very similar to the Sun 3/280 uniprocessor system that the simulations were actually run on. 64 processors is well beyond the maximum capacity of the bus used; the results obtained will show that the peak performance for this class of systems is obtained by using 33 processors. The processor speed, cache size, bus width, and cache line size were all chosen to be the same as those for the Sun 3/280. The three bus cycles to fetch or write back a cache line are based on the assumptions of an 8 byte bus with multiplexed address and data signals and a 16 byte cache line size. To transmit a cache line to or from main memory, three bus cycles are required; one for the address, and two to transmit the data in the line. The main memory access time of 160 ns is typical for currently available dynamic RAMs. Note that this time is not just the access time of the DRAM chips; it must also include the time for address decoding and other delays in the memory controller. The 14 ns bus transceiver delay is the delay for a pair of typical F series TTL bus transceiver devices. Each transceiver has a 7 ns delay, and a bus transaction must pass through two transceivers, one to get onto the bus and one to get off of it. The average of 6 CPU clocks between memory references by the CPU chip is derived from 68020 performance figures published in [MR85]. The cache miss ratio of 0.01486 and the figure of 34.96% for the fraction of misses that required a write back were obtained directly from the simulation workload by running the address trace through a simulation of the cache. The trace consisted of 1,775,947 memory references, of which 1,749,565 were cache hits and 26,382 were cache misses. Thus, the cache miss ratio is $\frac{26382}{1775947} = 0.01486$ for this workload. Of the cache misses, 9223 required writing a dirty line back to main memory. Thus, the fraction of misses requiring a write back operation was $\frac{9223}{26382} = 34.96\%$. Finally, the linear bus delay constant of 3.34 ns is the VME bus estimate derived in equation (3.10) in Chapter 3 from VME bus performance figures published in [Borri85].

4.4.1 Markov chain model results for 68020 example

From the above assumptions for the example system, parameters for the Markov chain model may be derived. The fixed time between requests excluding bus delay, t_r, can be obtained as follows:

$$t_r = \frac{\frac{240\,\text{ns}}{0.01486} + 160\,\text{ns} + 14\,\text{ns}}{3\ +\ (3)(0.3496)} = 4.033\,\mu\text{s}$$

The first term in the numerator is the mean processor compute time between references divided by the cache miss ratio, which gives the mean time between references from the

N	Bus Util.	Perfor-mance
1	0.01	0.99
2	0.01	1.99
3	0.02	2.98
4	0.03	3.97
5	0.04	4.96
6	0.06	5.94
7	0.07	6.93
8	0.09	7.91
9	0.10	8.89
10	0.12	9.87
11	0.15	10.84
12	0.17	11.81
13	0.19	12.78
14	0.22	13.75
15	0.25	14.71
16	0.28	15.66
17	0.31	16.61
18	0.34	17.56
19	0.37	18.49
20	0.41	19.42
21	0.44	20.33
22	0.48	21.23
23	0.52	22.11
24	0.56	22.97
25	0.60	23.79
26	0.64	24.58
27	0.68	25.33
28	0.72	26.01
29	0.75	26.62
30	0.79	27.15
31	0.83	27.56
32	0.86	27.86
33	0.89	28.04
34	0.91	28.08
35	0.93	28.00
36	0.95	27.81
37	0.96	27.53
38	0.97	27.18
39	0.98	26.77
40	0.99	26.34
41	0.99	25.88
42	0.99	25.40
43	1.00	24.93
44	1.00	24.46
45	1.00	24.00
46	1.00	23.55
47	1.00	23.11
48	1.00	22.68
49	1.00	22.26
50	1.00	21.86
51	1.00	21.48
52	1.00	21.10
53	1.00	20.74
54	1.00	20.39
55	1.00	20.05
56	1.00	19.72
57	1.00	19.41
58	1.00	19.10
59	1.00	18.80
60	1.00	18.51
61	1.00	18.23
62	1.00	17.96
63	1.00	17.70
64	1.00	17.44

Table 4.3: Markov chain model results for 68020 workload

cache. The second and third terms in the numerator are, respectively, the main memory access time and the bus transceiver delay. The denominator is the mean number of bus cycles for a single memory reference and is the sum of the number of bus cycles needed to service a cache miss (three) and the average number of cycles to write back dirty lines. Using this value of t_r to compute r_{lin}, we get

$$r_{\text{lin}} = \frac{k_{\text{lin}}}{t_r} = \frac{3.34\,ns}{4.033\,\mu\text{s}} = 0.0008285$$

The Markov chain model was used with this value of r_{lin} to determine the throughput for this architecture with N in the range 1 to 64 processors. Table 4.3 shows these results.

4.4.2 Trace driven simulation results for 68020 example

A trace driven simulation was then run for this example system. The results are shown in Table 4.4. For this system, the six MC68020 traces were concatenated to provide a single large job. Multiple instances of this job were run, one per processor, with each processor executing independently of the others. The figure obtained for total performance was defined to be the total number of memory references made by all of the processors divided by the total number of references that could be made in the same time by a single processor of the same type running the same workload.

N	Proc. Util.	Bus Util.	Mem. Util.	Performance
1	0.98	0.01	0.01	0.99
2	0.98	0.01	0.03	1.98
3	0.98	0.02	0.04	2.97
4	0.98	0.03	0.05	3.96
5	0.98	0.04	0.07	4.94
6	0.98	0.05	0.08	5.92
7	0.98	0.07	0.09	6.90
8	0.98	0.09	0.10	7.88
9	0.97	0.10	0.12	8.86
10	0.97	0.12	0.13	9.83
11	0.97	0.14	0.14	10.80
12	0.97	0.17	0.16	11.77
13	0.97	0.19	0.17	12.74
14	0.97	0.22	0.18	13.70
15	0.97	0.25	0.19	14.65
16	0.97	0.27	0.21	15.60
17	0.96	0.30	0.22	16.56
18	0.96	0.34	0.23	17.50
19	0.96	0.37	0.24	18.44
20	0.96	0.40	0.26	19.38
21	0.96	0.44	0.27	20.30
22	0.95	0.48	0.28	21.21
23	0.95	0.52	0.29	22.10
24	0.95	0.56	0.30	22.99
25	0.95	0.60	0.32	23.87
26	0.94	0.64	0.33	24.70
27	0.94	0.68	0.34	25.53
28	0.93	0.73	0.35	26.34
29	0.93	0.77	0.36	27.06
30	0.92	0.81	0.37	27.69
31	0.91	0.85	0.38	28.30
32	0.89	0.89	0.38	28.76
33	0.87	0.93	0.39	29.11
34	0.85	0.96	0.39	29.07
35	0.82	0.98	0.39	28.90
36	0.78	0.99	0.38	28.44
37	0.75	0.99	0.38	27.86
38	0.71	1.00	0.37	27.20
39	0.68	1.00	0.36	26.61
40	0.64	1.00	0.35	25.99
41	0.61	1.00	0.35	25.43
42	0.59	1.00	0.34	24.88
43	0.56	1.00	0.33	24.36
44	0.54	1.00	0.32	23.87
45	0.52	1.00	0.32	23.40
46	0.49	1.00	0.31	22.96
47	0.47	1.00	0.31	22.52
48	0.46	1.00	0.30	22.10
49	0.44	1.00	0.29	21.71
50	0.42	1.00	0.29	21.33
51	0.41	1.00	0.28	20.96
52	0.39	1.00	0.28	20.60
53	0.38	1.00	0.27	20.26
54	0.37	1.00	0.27	19.93
55	0.35	1.00	0.27	19.61
56	0.34	1.00	0.26	19.30
57	0.33	1.00	0.26	18.99
58	0.32	1.00	0.25	18.71
59	0.31	1.00	0.25	18.42
60	0.30	1.00	0.24	18.15
61	0.29	1.00	0.24	17.88
62	0.28	1.00	0.24	17.62
63	0.27	1.00	0.23	17.37
64	0.27	1.00	0.23	17.12

Table 4.4: Trace driven simulation results for 68020 workload

4.4.3 Accuracy of model for 68020 example

In this section the results obtained from the Markov chain model are compared with the results obtained from the trace driven simulation for the 68020 based example system. Table 4.5 shows both sets of results, along with the error percentage for the model with respect to the simulation.

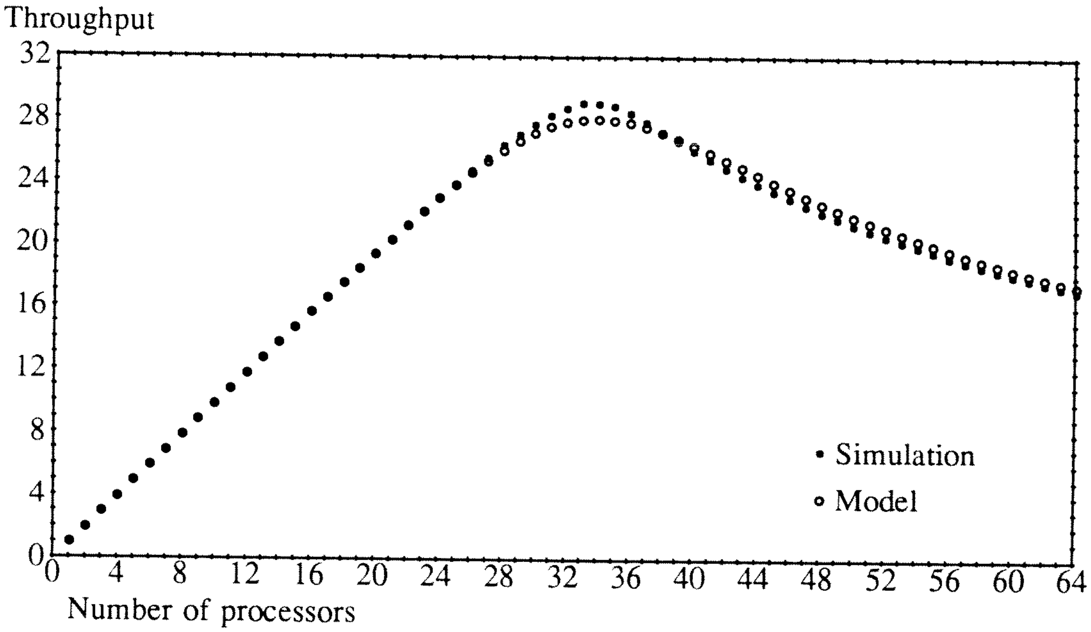


Figure 4.1: Comparison of results from 68020 workload

These values are plotted in Figure 4.1. From the figure, it can be seen that the results of the model and the simulation agree closely.

N	Model Result	Simulation Result	Percent Error
1	0.99	1.00	−0.13
2	1.99	1.99	−0.14
3	2.98	2.98	−0.16
4	3.97	3.98	−0.17
5	4.96	4.97	−0.19
6	5.94	5.96	−0.20
7	6.93	6.94	−0.24
8	7.91	7.93	−0.22
9	8.89	8.91	−0.24
10	9.87	9.89	−0.26
11	10.84	10.87	−0.30
12	11.81	11.84	−0.26
13	12.78	12.82	−0.29
14	13.75	13.79	−0.30
15	14.71	14.75	−0.26
16	15.66	15.70	−0.23
17	16.61	16.66	−0.25
18	17.56	17.60	−0.24
19	18.49	18.55	−0.32
20	19.42	19.49	−0.35
21	20.33	20.42	−0.41
22	21.23	21.33	−0.45
23	22.11	22.22	−0.52
24	22.97	23.10	−0.58
25	23.79	23.99	−0.81
26	24.58	24.81	−0.93
27	25.33	25.63	−1.20
28	26.01	26.43	−1.59
29	26.62	27.13	−1.86
30	27.15	27.77	−2.24
31	27.56	28.35	−2.77
32	27.86	28.78	−3.17
33	28.04	29.11	−3.69
34	28.08	29.07	−3.40
35	28.00	28.90	−3.11
36	27.81	28.41	−2.10
37	27.53	27.85	−1.16
38	27.18	27.17	+0.02
39	26.77	26.59	+0.69
40	26.34	25.97	+1.40
41	25.88	25.41	+1.84
42	25.40	24.86	+2.18
43	24.93	24.36	+2.36
44	24.46	23.86	+2.54
45	24.00	23.39	+2.62
46	23.55	22.95	+2.61
47	23.11	22.50	+2.67
48	22.68	22.08	+2.69
49	22.26	21.70	+2.59
50	21.86	21.33	+2.52
51	21.48	20.95	+2.52
52	21.10	20.59	+2.48
53	20.74	20.25	+2.40
54	20.39	19.93	+2.32
55	20.05	19.60	+2.29
56	19.72	19.29	+2.23
57	19.41	18.99	+2.21
58	19.10	18.70	+2.13
59	18.80	18.42	+2.07
60	18.51	18.14	+2.04
61	18.23	17.88	+1.99
62	17.96	17.62	+1.94
63	17.70	17.37	+1.90
64	17.44	17.13	+1.86

Table 4.5: Comparison of results from 68020 workload

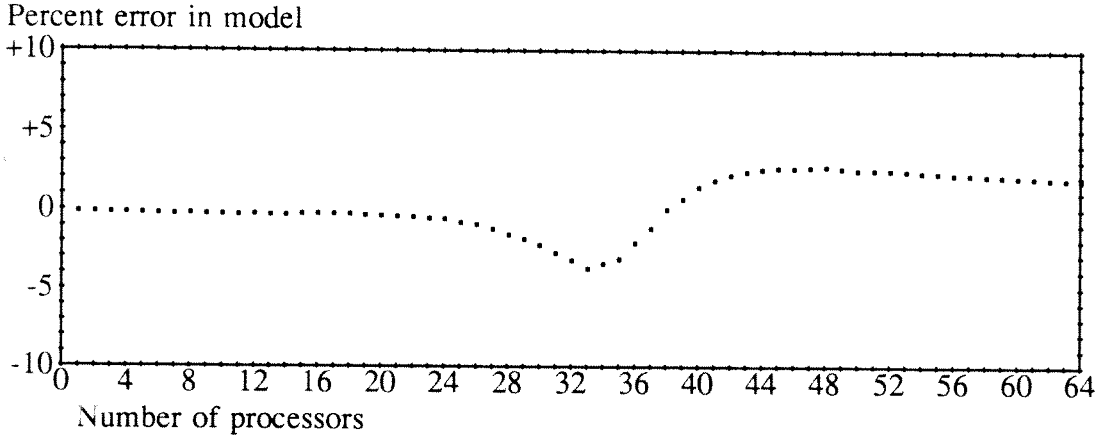


Figure 4.2: Percentage error in model for 68020 workload

The percentage differences between the two results are plotted in Figure 4.2. Note that the maximum discrepancy between the two results is approximately 3.7 percent. This compares favorably with the results of previous analytic models such as [Strec70, Bhand75, BS76, Hooge77, LVA82, MHBW84. MHBW86, MHW87] in which maximum errors ranging from 5.4 to 10.2 percent were observed. Other analytic performance models [GA84, Towsl86] have been reported to agree even more closely with simulation results. However, all of these results except for [Hooge77] were based on simulated reference patterns derived from random number generators and not actual program data, and the traces used in [Hooge77] consisted of only ten thousand memory references. We have demonstrated that the performance estimates from our model agree closely with the performance figures obtained using long traces (1.77 million memory references) from actual programs.

4.5 Results for 88100 example system

The example system using the 88100 processor was designed based on the following assumptions:

- The system contains 1 to 16 processors.
- Each processor is a Motorola MC88100 running at 25 MHz.
- Each processor has a 2 Kbyte cache.
- The bus data path width is 4 bytes.
- The cache line size is 16 bytes.
- 9 bus cycles are needed to fetch data for a cache miss.
- 9 bus cycles are needed to write a dirty line back from the cache.
- The bus runs synchronously with the processors at 25 MHz.

The example problem used a 64×64 matrix. This size was chosen because it was the largest one that allowed the simulations to complete in a reasonable time on the Sun 4/280 system used to run the simulator. A 64×64 matrix of double precision (8 byte) elements takes $64 \times 64 \times 8 = 32768$ bytes of storage. This is small enough to fit into the 128 Kbyte direct mapped cache on the Sun 4/280 processor along with most of the simulator program. Larger matrices tend to conflict with the simulator for the use of the cache on the Sun, resulting in a large increase in simulator execution time.

The cache size selected for the example system, 2 Kbytes, is small by current standards. This small size was selected to ensure that the example problem did not entirely fit in the caches in the simulated multiprocessor. When the problem fits entirely in the caches, the bus traffic is extremely low, consisting only of those operations needed to fill the caches or to exchange shared data. This results in a system in which the overall performance is relatively insensitive to the bus performance. We chose to consider the case in which the cache miss ratio is significant, to allow the effects of bus performance on overall system performance to be investigated. The results obtained in this way should be indicative of the performance of larger systems running larger problems.

For the selected workload, the bus request rate per processor is dependent on the number of processors. This differs from the 68020 workload, in which the request rate is independent of the number of processors. The major reason for the dependency of the reference rate on the number of processors is that the total amount of cache memory in the system is proportional to the number of processors but the problem size is constant. As a consequence, the cache miss rate decreases as the number of processors is increased, thus decreasing the bus request rate. For the 68020 workload, on the other hand, each processor is executing an independent process, so each process only has a single cache memory. Another reason for the dependency is that the 88100 workload represents a true parallel algorithm, and the sequence of references from a particular processor depends on the operations of the other processors.

In Table 4.6, the miss ratio and the mean number of clock cycles of compute time per bus request are shown as a function of the number of processors. It can be seen that the miss ratio steadily decreases and the time between requests steadily increases as the number of processors increases. This is due to the increase in the total available cache memory described previously.

N	Miss ratio	Clocks per bus request
1	0.0148	30.6
2	0.0138	33.0
3	0.0127	36.1
4	0.0118	39.4
5	0.0107	43.8
6	0.0089	52.9
7	0.0077	61.5
8	0.0072	66.7
9	0.0069	70.7
10	0.0067	73.7
11	0.0064	78.0
12	0.0059	84.4
13	0.0057	89.4
14	0.0056	91.5
15	0.0053	98.5
16	0.0048	109.5

Table 4.6: Clocks per bus request for 88100 workload

4.5.1 Markov chain model results for 88100 example

To apply the Markov chain model to this system, the value of v, the mean compute time per bus cycle, is needed for each system configuration to be analyzed. These values may be obtained by dividing the request rate values given in Table 4.6 by the number of clock cycles per bus operation, which is always 9 for this example. Given this value of v and the number of processors N, an iterative solution to equations (3.2) and (3.5) may be obtained. From this solution, the bus utilization may be obtained using equation (3.6). The Markov chain model estimates for the mean service time s and the bus utilization U for the example system and workload are shown in Table 4.7.

N	Request rate	v	Service time s	Bus Util. U
1	30.6	3.40	1.00	0.23
2	33.0	3.67	1.05	0.41
3	36.1	4.01	1.16	0.55
4	39.4	4.37	1.30	0.65
5	43.8	4.86	1.45	0.72
6	52.9	5.88	1.54	0.74
7	61.5	6.84	1.62	0.75
8	66.7	7.41	1.77	0.79
9	70.7	7.86	1.95	0.82
10	73.7	8.18	2.17	0.85
11	78.0	8.67	2.36	0.87
12	84.4	9.38	2.50	0.88
13	89.4	9.94	2.68	0.90
14	91.5	10.16	2.97	0.92
15	98.5	10.94	3.09	0.92
16	109.5	12.16	3.06	0.92

Table 4.7: Markov chain model results for 88100 workload

4.5.2 Trace driven simulation results for 88100 example

Complete simulations were then run for this example system. The results are shown in Table 4.8.

N	88100 time (ms.)	Speedup over $N = 1$	Bus utilization	Mean service time	Simulator time (minutes)
1	122.9	1.0	0.23	1.04	8.3
2	67.3	1.8	0.42	1.12	9.0
3	49.9	2.5	0.57	1.25	9.9
4	41.7	2.9	0.69	1.44	11.0
5	38.0	3.2	0.76	1.69	12.6
6	38.5	3.2	0.76	2.05	16.0
7	38.7	3.2	0.76	2.37	18.2
8	37.3	3.3	0.79	2.68	20.0
9	36.5	3.4	0.82	3.16	23.5
10	35.7	3.4	0.84	3.76	24.1
11	34.7	3.5	0.86	4.08	26.3
12	35.0	3.5	0.86	4.64	27.2
13	34.4	3.6	0.87	5.01	29.2
14	33.8	3.6	0.88	5.80	30.9
15	33.7	3.6	0.88	6.16	34.1
16	33.8	3.6	0.87	6.12	36.2

Table 4.8: Trace driven simulation results for 88100 workload

It can be seen that the use of multiple processors can result in a significant speedup for this example problem (for $N \leq 4$). The speedup as a function of the number of processors is plotted in Figure 4.3. The small value for the maximum speedup (3.6) is due to the small cache size. With larger caches, significantly better speedups can be obtained. For example, by increasing the cache size to 16 Kbytes per processor, a maximum speedup of 9.5 can be obtained by using 22 processors.

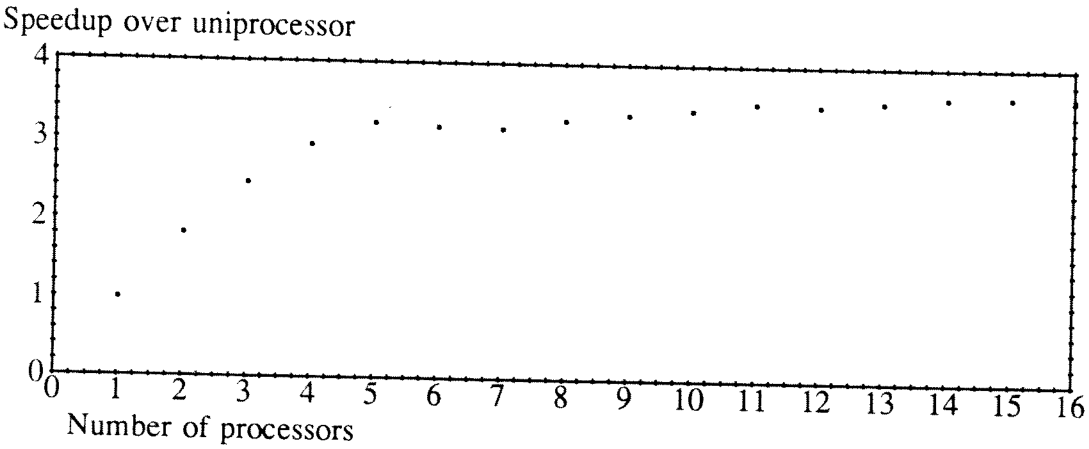


Figure 4.3: Speedup for parallel **dgefa** algorithm

4.5.3 Accuracy of model for 88100 example

In this section the results obtained from the Markov chain model are compared with the results obtained from the trace driven simulation for the 88100 based example system. Table 4.9 shows both sets of results, along with the error percentages for the model with respect to the simulation. From these results, it is apparent that the model does not provide an accurate estimate of service time when the independence assumption is violated. It does, however, still provide an estimate of bus utilization that is accurate to within 6 percent. The bus utilization for both the simulation and the model are plotted in Figure 4.4. The error percentage for the bus utilization predicted by the model is plotted in Figure 4.5.

4.6 Summary of results for single logical bus

This section summarizes the results for systems constructed with a single logical bus. We have developed a new Markov chain model for multiprocessor performance

	Simulation		Model		Bus	Service
	Bus	Service	Bus	Service	util.	time
N	util.	time	util.	time	% error	% error
1	0.23	1.04	0.23	1.00	+0.91	− 3.9
2	0.42	1.12	0.41	1.05	−1.33	− 5.9
3	0.57	1.25	0.55	1.16	−3.41	− 7.8
4	0.69	1.44	0.65	1.30	−5.19	− 9.7
5	0.76	1.69	0.72	1.45	−5.57	−14.3
6	0.76	2.05	0.74	1.54	−2.63	−25.0
7	0.76	2.37	0.75	1.62	−0.77	−31.7
8	0.79	2.68	0.79	1.77	−0.58	−34.1
9	0.82	3.16	0.82	1.95	+0.51	−38.3
10	0.84	3.76	0.85	2.17	+1.82	−42.2
11	0.86	4.08	0.87	2.36	+1.21	−42.0
12	0.86	4.64	0.88	2.50	+3.20	−46.0
13	0.87	5.01	0.90	2.68	+3.10	−46.4
14	0.88	5.80	0.92	2.97	+4.48	−48.8
15	0.88	6.16	0.92	3.09	+4.96	−49.9
16	0.87	6.12	0.92	3.06	+4.59	−50.0

Table 4.9: Comparison of results from 88100 workload

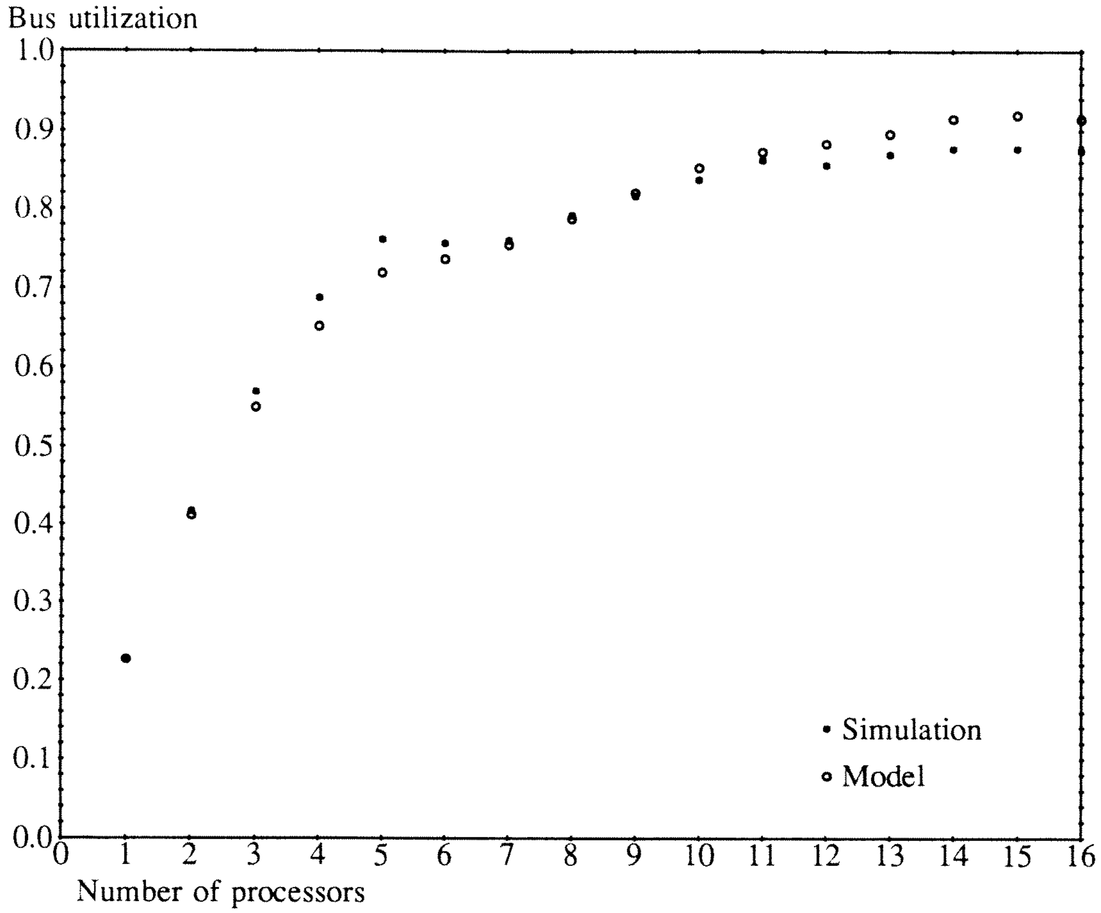


Figure 4.4: Comparison of results for 88100 workload

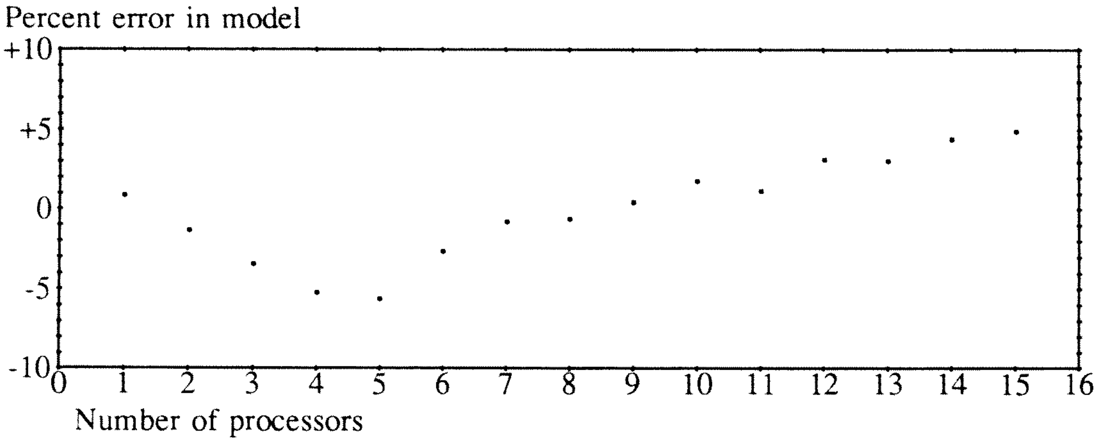


Figure 4.5: Percentage error in model for 88100 workload

that considers the effects of bus loading. Using trace driven simulations, we have demonstrated that this model provides very accurate results for systems in which the references from different processors are independent. For systems that lack this independence, our model is not well suited for estimating the bus service time, but it still provides a reasonably accurate estimate of bus utilization.

CHAPTER 5

CROSSPOINT CACHE ARCHITECTURE

In this chapter, we propose a new cache architecture that allows a multiple bus system to be constructed which assures hardware cache consistency while avoiding the performance bottlenecks associated with previous hardware solutions [MHW87]. Our proposed architecture is a crossbar interconnection network with a cache memory at each crosspoint [WM87]. Crossbars have traditionally been avoided because of their complexity. However, for the "non-square" systems that we are focusing on with 16 or 32 processors per memory bank and no more than 4 to 8 memory banks, the number of crosspoint switches required is not excessive. The simplicity and regular structure of the crossbar architecture greatly outweigh any disadvantages due to complexity. We will show that our architecture allows the use of straightforward and efficient bus oriented cache consistency schemes while overcoming their bus traffic limitations.

5.1 Single bus architecture

Figure 5.1 shows the architecture of a single bus multiprocessor with snooping caches. Each processor has a private cache memory. The caches all service their misses by going to main memory over the shared bus. Cache consistency is ensured by using a snooping cache protocol in which each cache monitors all bus addresses.

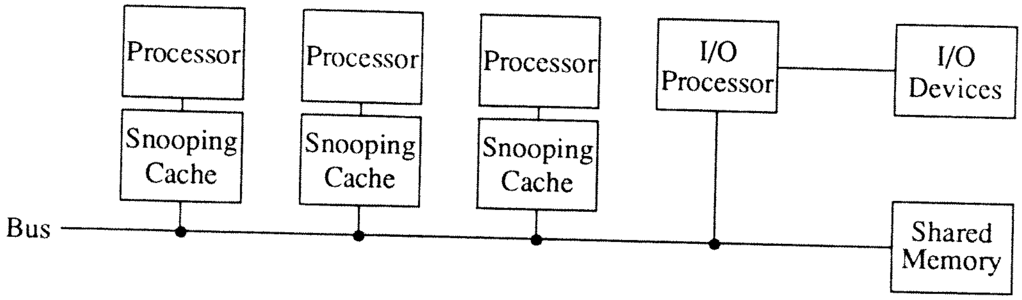


Figure 5.1: Single bus with snooping caches

Cache consistency problems due to input/output operations are solved by connecting the I/O processors to the shared bus and having them observe the bus cache consistency protocol. Since the only function of the I/O processor is to transfer data to and from main memory for use by other processors, there is little or no advantage in using cache memory with it. It may be desirable to cache disk blocks in main memory, but this is a software issue unrelated to the use of a hardware cache between the I/O processor and the bus.

Although this architecture is simple and inexpensive, the bandwidth of the shared bus severely limits its maximum performance. Furthermore, since the bus is shared by all the processors, arbitration logic is needed to control access to the bus. Logic delays in the arbitration circuitry may impose additional performance penalties.

5.2 Crossbar architecture

Figure 5.2 shows the architecture of a crossbar network. Each processor has its own bus, as does each memory bank. The processor and memory buses are oriented (at least conceptually) at right angles to each other, forming a two-dimensional grid. A crosspoint switch is placed at each intersection of a processor bus and a memory bus. Each crosspoint switch consists of a bidirectional bus transceiver and the control logic needed to enable the transceiver at the appropriate times. This array of crosspoint switches allows any processor to be connected to any memory bank through a single switching element. Arbitration is still needed on the memory buses, since each is shared by all processors. Thus, this architecture does not eliminate arbitration delay.

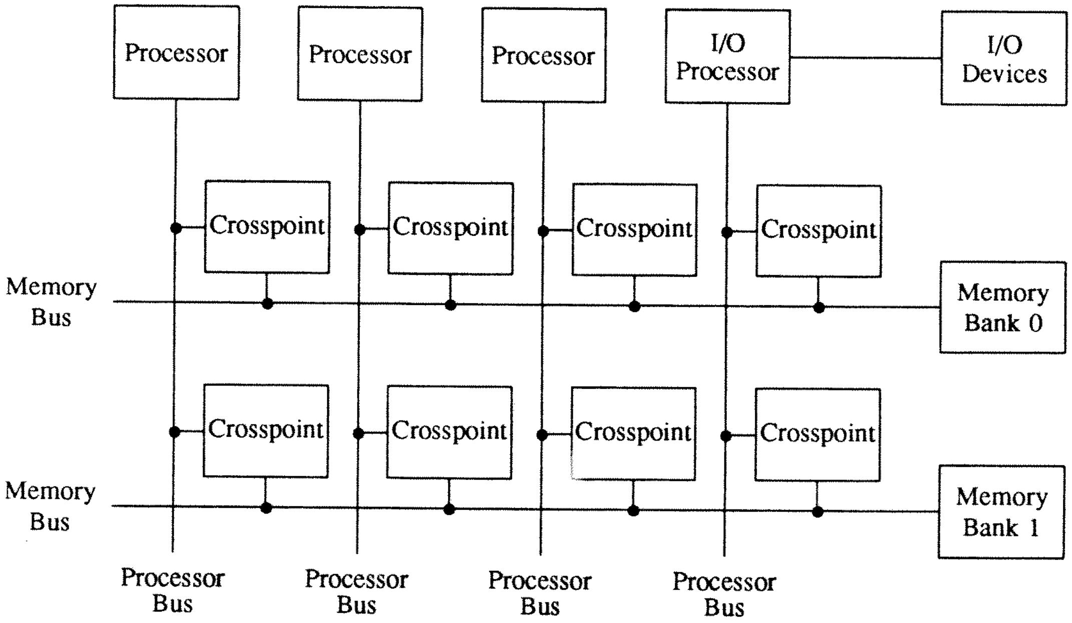


Figure 5.2: Crossbar network

The crossbar architecture is more expensive than a single bus. However, it avoids the performance bottleneck of the single bus, since several memory requests may be serviced simultaneously. Unfortunately, if a cache were associated with each processor in this architecture, as shown in Figure 5.3, cache consistency would be difficult to achieve. The snooping cache schemes would not work, since there is no reasonable way for every processor to monitor all the memory references of every other processor.

Each processor would have to monitor activity on every memory bus simultaneously. To overcome this problem, we propose the crosspoint cache architecture.

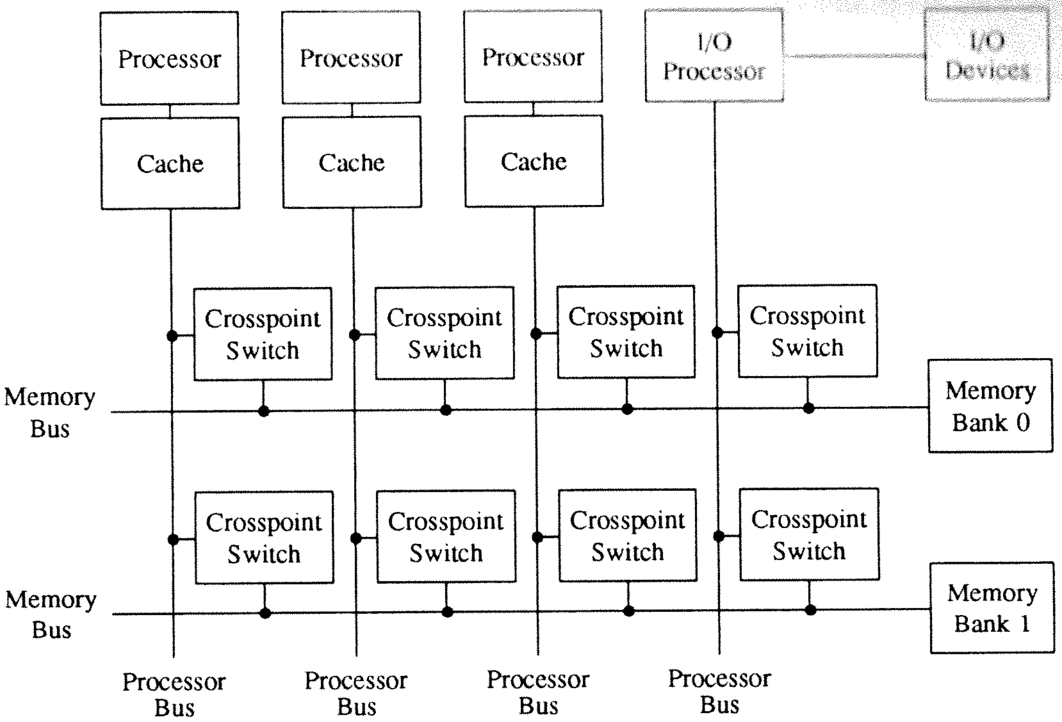


Figure 5.3: Crossbar network with caches

5.3 Crosspoint cache architecture

In the crosspoint cache architecture, the general structure is similar to that of the crossbar network shown in Figure 5.2, with the addition of a cache memory in each crosspoint. This architecture is shown in Figure 5.4.

For each processor, the multiple crosspoint cache memories that serve it (those attached to its processor bus) behave similarly to a larger single cache memory. For example, in a system with four memory banks and a 16K byte direct mapped cache

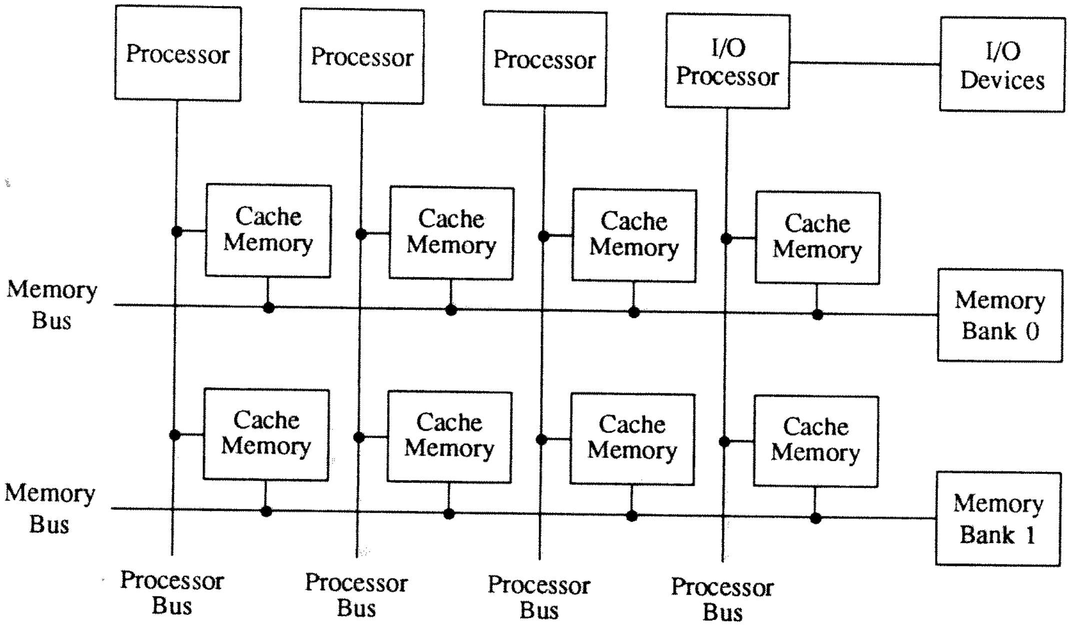


Figure 5.4: Crosspoint cache architecture

with a 16 byte line size at each crosspoint, each processor would “see” a single 64K byte direct mapped cache with a 16 byte line size. Note that this use of multiple caches with each processor increases the total cache size, but it does not affect the line size or the degree of set associativity. This approach is, in effect, an interleaving of the entire memory subsystem, including both the caches and the main memory.

To explain the detailed functioning of this system, we consider processor bus activity and memory bus activity separately.

5.3.1 Processor bus activity

Each processor has the exclusive use of its processor bus and all the caches connected to it. There is only one cache in which a memory reference of a particular processor to a particular memory bank may be cached. This is the cache at the intersection of the corresponding processor and memory buses.

The processor bus bandwidth requirement is low, since each bus needs only enough bandwidth to service the memory requests of a single processor. The cache bandwidth requirement is even lower, since each cache only handles requests from a single processor, and it only services those requests directed to a particular memory bank.

Note that this is not a shared cache system. Since each processor bus and the caches on it are dedicated to a single processor, arbitration is not needed for a processor bus or its caches. Furthermore, bus interference and cache interference cannot occur on the processor buses, since only the processor can initiate requests on these buses. Thus, the principal delays associated with shared cache systems are avoided.

5.3.2 Memory bus activity

When a cache miss occurs, a memory bus transaction is necessary. The cache that missed places the requested memory address on the bus and waits for main memory (or sometimes another cache) to supply the data. Since all the caches on a particular memory bus may generate bus requests, bus arbitration is necessary on the memory buses. Also, since data from a particular memory bank may be cached in any of the caches connected to the corresponding memory bus, it is necessary to observe a cache consistency protocol along the memory buses. The cache consistency protocol will make memory bus operations necessary for write hits to shared lines as well as for cache misses.

Since each memory bus services only a fraction of each processor's cache misses, this architecture can support more processors than a single bus system before reaching the upper bound on performance imposed by the memory bus bandwidth. For example, if main memory were divided into four banks, each with its own memory bus, then each memory bus would only service an average of one fourth of all the cache misses in the system. So, the memory bus bandwidth would allow four times as many processors as a single bus snooping cache system.

Also note that unlike the multiple bus architecture described in Chapter 2, no B-of-M arbiter is needed. Since each memory bus is dedicated to a specific memory bank, a simple 1-of-N arbiter for each memory bus will suffice.

5.3.3 Memory addressing example

To better illustrate the memory addressing in the crosspoint cache architecture, we consider a system with the following parameters: 64 processors, 4 memory banks, 256 crosspoint caches, 32-bit byte addressable address space, 32-bit word size, 32-bit bus width, 4 word (16 byte) crosspoint cache line size, 16K byte (1024 lines) crosspoint cache size, and direct mapped crosspoint caches.

When a processor issues a memory request, the 32 bits of the memory address are used as follows: The two least significant bits select one of the four bytes in a word. The next two bits select one of the four words in a cache line. The next two bits select one of the four memory banks, and thus one of the four crosspoint caches associated with the requesting processor. The next ten bits select one of the 1024 lines in a particular crosspoint cache. The remaining bits (the most significant 16) are the ones compared with the address tag in the crosspoint cache to see if a cache hit occurs. This address bit mapping is illustrated in Figure 5.5.

MSB		Address bits		LSB
16	10	2	2	2
Tag	Line in cache	Memory bank select	Word in line	Byte in word

Figure 5.5: Address bit mapping example

5.4 Performance considerations

To estimate the performance of a system using the crosspoint cache architecture, the following approach may be used. First, the rate at which each crosspoint cache generates memory requests is needed. This may be obtained by treating all of the crosspoint caches associated with one of the processors as if it were a single cache equal to the sum of all the crosspoint cache sizes. The rate at which this hypothetical large cache generates requests is computed by multiplying its expected miss ratio by the memory request rate of the processor. The request rate from this large cache is then divided by the number of crosspoint caches per processor, since we assume that interleaving distributes requests evenly among the memory buses. Then, given the request rate from each of the crosspoint caches, the Markov chain model developed in Chapter 3 may be used to estimate the performance of the memory buses. Multiplying the memory bus traffic figures thus obtained by the number of memory buses will give the total rate at which data is delivered from main memory. Dividing this figure by the cache miss ratio will give the total rate at which data is delivered to the processors. This may be used directly as a measure of total system throughput.

Specifically, in Chapter 3 it was found that for a single level linear bus,

$$N_{\max} \approx \sqrt{\frac{t_r}{k_{\text{lin}}}} = \sqrt{\frac{1}{r_{\text{lin}}}}$$

When the crosspoint cache architecture is used, memory references are divided among the memory modules. Thus, for a particular bus, t_r, the mean time between requests from a particular processor is increased by a factor equal to the number of memory modules. Adapting the result from Chapter 3 for a crosspoint cache system with M memory modules, we get

$$N_{\max} \approx \sqrt{\frac{Mt_r}{k_{\text{lin}}}} = \sqrt{\frac{M}{r_{\text{lin}}}} \tag{5.1}$$

From this, it can be seen that an increase in the maximum number of processors by a factor of $\sqrt{2} \approx 1.414$ can be obtained by changing a single bus system to a crosspoint cache system with two memory modules. Similarly, increases in the maximum number of processors by factors of 2 and 2.828 can be obtained by using four and eight buses, respectively.

We can also determine the necessary number of memory modules for particular values of N and r_{lin}. From equation (5.1) we get

$$M \approx N^2 r_{\text{lin}}$$

Bus loading on the memory buses may limit the maximum size of crosspoint cache systems, since all the processors must be connected to each memory bus. As the number of processors becomes large, bus propagation delays and capacitive loading will reduce the maximum speed and bandwidth of the bus. However, with cache memories, the impact of a slower bus on system performance will be reduced, since most memory references will be satisfied by the caches and will not involve the bus at all. Similarly, the bus traffic reduction obtained from the caches will offset the reduced bandwidth of a slower bus. Furthermore, the hierarchical bus organizations investigated in Chapter 3 may be used to construct the memory buses. This approach will be considered in detail in Chapter 6.

5.5 Two-level caches

The performance of the crosspoint cache architecture may be further improved by adding a local cache between each processor and its processor bus. To see why this is so, we examine some of the tradeoffs in designing a crosspoint cache system.

Since memory bus bandwidth is a critical resource, large crosspoint caches are desirable to maximize the cache hit rate, thus minimizing the memory bus traffic. Cache speed is one of the most important factors influencing a processor's average memory access time. Thus, the speed of the crosspoint caches should also be as fast as possible to maximize the performance of each individual processor. Simultaneously achieving the goals of low bus traffic and fast memory access would be expensive, though, since large amounts of fast memory would be necessary.

Two of the major performance limitations of multiple bus systems are reduced or eliminated by the crosspoint cache architecture: arbitration delays and cache interference. Arbitration delays on the processor buses are eliminated altogether; since only the processor can initiate requests on the bus, no arbitration is required. Arbitration delays on the memory buses are greatly reduced, since a simple, fast 1-of-N arbiter can be used instead of a complex, slow B-of-M arbiter. Cache interference occurs when a processor is blocked from accessing its cache because the cache is busy servicing another request, typically a snooping operation from the bus. In a standard multiple bus system, an operation on any of the buses can require service from a cache; thus, the fraction of time that a cache is busy and unavailable for its processor may be large. With the crosspoint cache architecture, on the other hand, only a single memory bus can require service from a particular cache. This reduces the frequency with which processors are blocked from accessing their caches because the cache is busy.

The principal remaining performance limitation in the crosspoint cache architecture is the processor bus delay. Bus propagation delays, capacitive loading, and delays from the bus interface logic limit the maximum feasible speed of this bus. To address this problem, we propose the addition of a second level of cache memories between the processors and their respective processor buses. This modified crosspoint cache architecture is examined in the following section.

5.5.1 Two-level crosspoint cache architecture

By placing a fast cache between each processor and its processor bus, the effect of the processor bus and crosspoint cache delays can be greatly reduced. When speed is the primary consideration, the best possible location for a processor's cache is on the processor chip itself. On-chip caches can be extremely fast, since they avoid the delays due to IC packaging and circuit board wiring. They are limited to a small size, however, since the limited area of a microprocessor chip must be allocated to the processor itself, the cache, and any other special features or performance enhancements desired. By combining a fast but small on-chip cache with large but slow crosspoint caches, the benefits of both can be realized. The fast on-chip cache will serve primarily to keep average memory access time small, while the large crosspoint caches will keep memory bus traffic low. This architecture with two cache levels is shown in Figure 5.6.

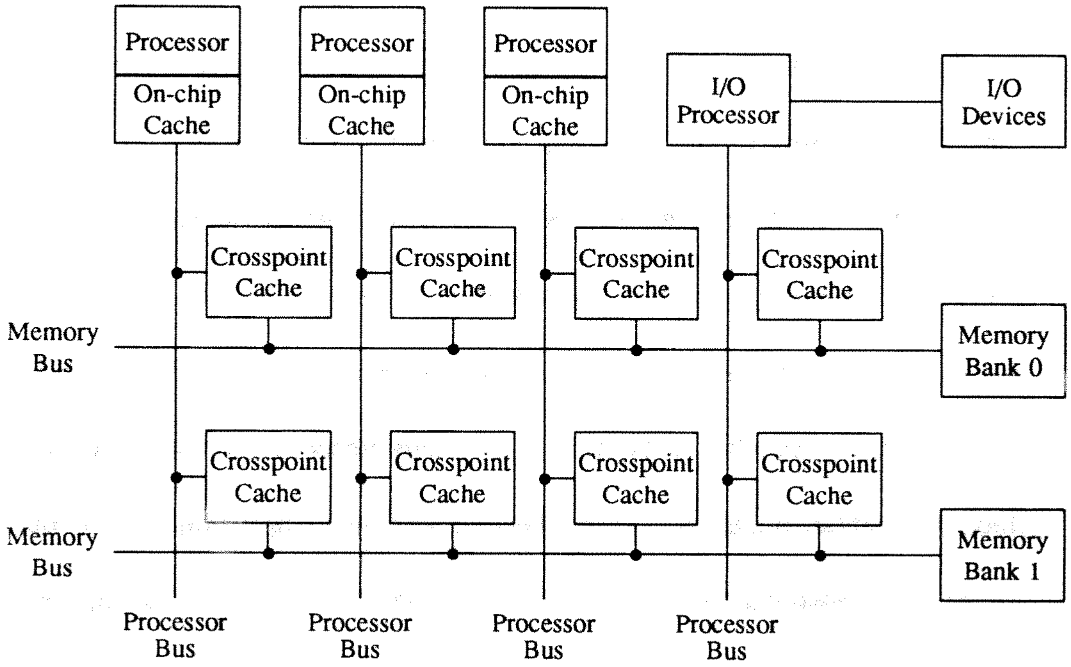


Figure 5.6: Crosspoint cache architecture with two cache levels

5.5.2 Cache consistency with two-level caches

Using a two-level cache scheme introduces additional cache consistency problems. Fortunately, a simple solution is possible.

In our cache consistency solution, a snooping protocol such as Illinois, Berkeley, or EDWP is used on the memory buses to ensure consistency between the crosspoint caches. The on-chip caches use write through to ensure that the crosspoint caches always have current data. The high traffic of write through caches is not a problem, since the processor buses are only used by a single processor. Since write through ensures that lines in the crosspoint caches are always current, the crosspoint caches can service any read references to shared lines that they contain without interfering with the processor or its on-chip cache.

Special attention must be given to the case in which a processor writes to a shared line that is present in another processor's on-chip cache. It is undesirable to send all shared writes to the on-chip caches since this would reduce the bandwidth of the on-chip caches that is available to their processors.

If each crosspoint cache can always determine whether one of its lines is also present in its associated on-chip cache, then it can restrict accesses to the on-chip cache to only those that are absolutely necessary. When a write to a shared line hits on the crosspoint cache, the crosspoint cache can send an invalidation request to the line in the on-chip cache only if the on-chip cache really has the line.

With suitable cache design, the crosspoint cache can determine whether one of its lines is currently in the on-chip cache, since the on-chip cache must go through the crosspoint cache to obtain all its lines. To see how this can be done, we consider the simplest case. This occurs when both the on-chip and crosspoint caches are direct mapped and have equal line sizes.

In a direct mapped cache, there is only a single location in which a particular line from main memory may be placed. In most designs, this location is selected by the bits of the memory address just above the bits used to select a particular word in the line. If the total size of the crosspoint caches is larger than that of their associated on-chip cache and all line sizes are equal, then the address bits that are used to select a particular crosspoint cache entry will be a superset of those bits used to select the on-chip cache entry. Consider those address bits that are used to select the crosspoint cache entry but not the on-chip cache entry. Out of a group of all crosspoint cache entries that differ only in these address bits, exactly one will be in the on-chip cache at any given time. If the value of these address bits is recorded in a special memory in the crosspoint caches for each line obtained by the on-chip cache, a complete record of which lines are in the on-chip cache will be available.

Note that it is not possible for a line to be in the on-chip cache but not the crosspoint cache. In other words, if a line is in the on-chip cache, it must be in the crosspoint cache. This important restriction has been called the *inclusion property* [BW88]. A sufficient condition for inclusion with direct mapped caches is stated in [WM87]. A detailed study of inclusion for several cache organizations is presented in [BW88]. Unfortunately, the authors of [BW88] appear to have misunderstood the results of [WM87].

To determine if a particular crosspoint cache line is in the on-chip cache, the crosspoint cache uses the same address bits used to select the on-chip cache entry to select an entry in this special memory. It then compares the additional address bits used to select the crosspoint cache entry with the value of those bits that is stored in the special memory. These bits will be equal if and only if the line is in the on-chip cache.

The size in bits of the special memory for each crosspoint cache is given by

$$\left(\frac{L_{oc}}{M}\right) \log_2 \left(M \frac{L_{xp}}{L_{oc}}\right)$$

where L_{oc} is the number of lines in the on-chip cache, L_{xp} is the number of lines in each crosspoint cache, and M is the number of memory banks. This is not a large amount of memory. Consider our example system with four memory banks, a line size of 16 bytes, and a crosspoint cache size of 16K bytes. We will assume a 1K byte on-chip cache is added to each processor. Figure 5.7 illustrates the addressing for this example. In this case, we have $M = 4$, $L_{xp} = 1024$, and $L_{oc} = 64$, so only 96 bits per crosspoint cache are needed to keep track of the lines in the on-chip cache.

	MSB		Address bits			LSB
	16	6	4	2	2	2
On-chip cache bit mapping	Tag for on-chip cache		Line in on-chip cache		Word in line	Byte in word
Crosspoint cache bit mapping	Tag for crosspoint cache	Line in crosspoint cache		Memory bank select	Word in line	Byte in word

Figure 5.7: Address bit mapping example for two cache levels

This approach is more difficult to use with set associative on-chip caches since additional signals must be provided on the microprocessor to allow the on-chip cache to inform the crosspoint caches of the location (which element in the set) of each line it loads. In future systems, however, direct mapped caches are likely to see more frequent use than set associative caches, since direct mapped caches are significantly faster for large cache sizes [Hill87, Hill88].

A disadvantage of this two-level cache consistency approach is that it requires arbitration on the processor buses, since the crosspoint caches use these buses to issue the invalidation requests. This will decrease the effective speed of the processor buses, so the time required to service a miss for the on-chip cache will be slightly greater than the memory access time of a similar system without the on-chip caches.

5.6 VLSI implementation considerations

Using VLSI technology to build a crossbar network requires an extremely large number of pin connections. For example, a crossbar network with 64 processors, 4 memory banks, and a 32 bit multiplexed address and data path requires at least 2208 connections. Present VLSI packaging technology is limited to a maximum of several hundred pins. Thus, a crossbar of this size must be partitioned across multiple packages.

The most straightforward partitioning of a crossbar network is to use one package per crosspoint. This results in a design that is simple and easy to expand to any desired size. The complexity of a single crosspoint is roughly equivalent to an MSI TTL package, so the ratio of pins to gates is high. This approach leads to MSI circuitry in VLSI packages, so it does not fully exploit the capabilities of VLSI. A much better pin to gate ratio is obtained by using a bit-sliced partitioning in which each package contains a single bit of the data path of the entire network. The bit-sliced approach, however, is difficult to expand since the network size is locked into the IC design.

The crosspoint cache architecture, on the other hand, permits the construction of a single VLSI component, which contains the crosspoint cache and its bus interfaces, that is efficient in systems spanning a wide performance range. If each package contains a single crosspoint cache, the number of pins required is reasonable, and the cache size may be made as large as necessary to take full advantage of the available silicon area. It also allows the same chip to be used both in small systems with just a few processors and a single memory bank and in large systems with a hundred or more processors and eight or sixteen memory banks.

In the example given, each crosspoint cache contains 128K bits of data storage, approximately 20K bits of tag storage, and some fairly simple switch and control logic. Since static RAMs as large as 256K bits are widely available, it should be feasible to construct such a crosspoint cache on a single chip with present VLSI technology.

5.7 Summary

To overcome the performance limitations of shared memory systems with a single bus while retaining many of their advantages, we have proposed the crosspoint cache architecture. We have shown that this architecture would permit shared memory multiprocessor systems to be constructed with more processors than present systems, while avoiding the need for the software enforcement of cache consistency.

We have also described a two-level cache architecture in which both crosspoint caches and caches on the processor chips are used. This architecture uses small but fast on-chip caches and large but slow crosspoint caches to achieve the goals of fast memory access and low bus traffic in a cost effective way.

A potential problem with this architecture is that the use of the multiple memory buses will permit the number of processors to increase to the point at which loading on the memory buses becomes a serious problem. This issued is addressed in the next chapter.

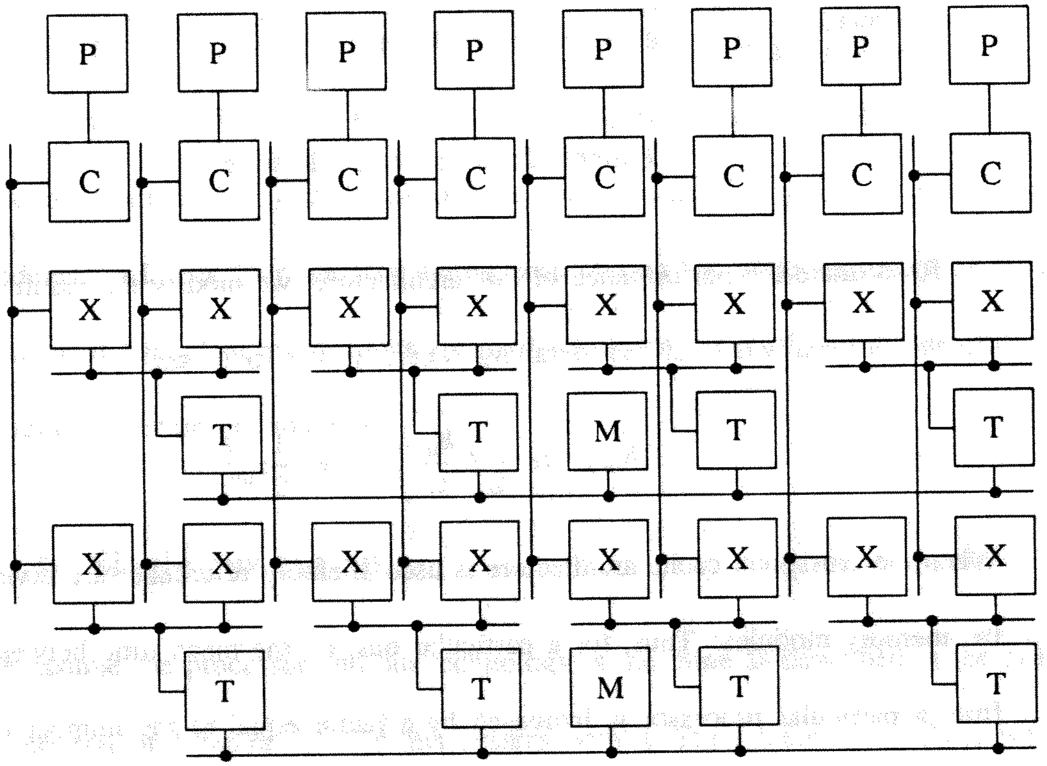
CHAPTER 6

LARGE SYSTEMS

In this chapter, we show how the crosspoint cache architecture described in Chapter 5 can be combined with the hierarchical bus designs investigated in Chapter 3 to construct large shared memory multiprocessor systems.

6.1 Crosspoint cache system with two-level buses

As discussed in Chapter 5, the principal problem of the crosspoint cache architecture is bus loading on the memory buses. Since the memory buses each have one connection per processor, the loading on these buses will significantly reduce their speed as the number of processors becomes large. To overcome this problem, we propose the use of two-level buses as described in Chapter 3 to implement the memory buses in a crosspoint cache system. The resulting architecture is shown in Figure 6.1. For readability, a very small system is shown in the figure with only 8 processors, and two memory modules. The two-level memory buses are organized to arrange the processors and their associated crosspoint caches into four groups with two processors in each group. A larger system of this design with 32 processors and four memory modules is shown in somewhat less detail in Figure 6.2. We would expect a practical system of this design to have from several dozen to several hundred processors.



P processor
C on-chip cache
X crosspoint cache
T bus to bus transceiver
M main memory module

Figure 6.1: Hierarchical bus crosspoint cache system

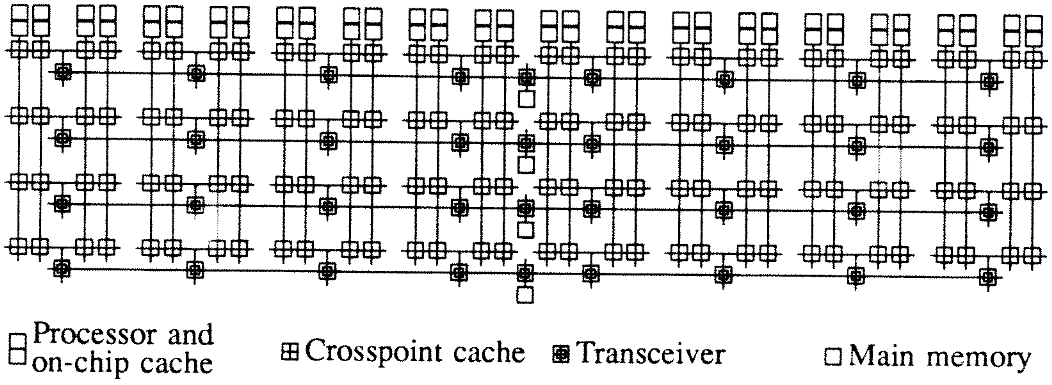


Figure 6.2: Larger example system

To estimate the performance of this architecture, we modify the result obtained in Chapter 3 for a two-level bus hierarchy. There, it was found that

$$N_{\mathrm{max}} \approx \frac{1}{2}\left(\frac{k_{\mathrm{lin}}}{t_r}\right)^{-\frac{2}{3}} = \frac{1}{2}r_{\mathrm{lin}}^{-\frac{2}{3}}$$

When the crosspoint cache architecture is used, memory references are divided among the memory modules. Thus, for a particular bus, t_r, the mean time between requests from a particular processor is increased by a factor equal to the number of memory modules. Adapting the result from Chapter 3 for a crosspoint cache system with M memory modules, each of which is connected to a two-level bus, we get

$$N_{\mathrm{max}} \approx \frac{1}{2}\left(\frac{k_{\mathrm{lin}}}{Mt_r}\right)^{-\frac{2}{3}} = \frac{1}{2}\left(\frac{r_{\mathrm{lin}}}{M}\right)^{-\frac{2}{3}} \qquad (6.1)$$

From this, it can be seen that an increase in the maximum number of processors by a factor of $\sqrt[3]{4} \approx 1.587$ can be obtained by changing a single bus system to a two-level bus crosspoint cache system with two memory modules. Similarly, increases in the maximum number of processors by factors of 2.520 and 4 can be obtained by using four and eight buses, respectively.

We can also determine the necessary number of memory modules for particular values of N and r_{lin}. From equation (6.1) we get

$$M \approx (2N)^{\frac{3}{2}} r_{\text{lin}}$$

6.2 Large crosspoint cache system examples

We conclude this chapter by extending the examples presented at the end of Chapter 3 to a crosspoint cache system with four memory modules. A system with four memory modules, and thus four memory buses, was chosen for these examples because the number of backplane connections required for this number of buses is feasible using current connector technology.

6.2.1 Single bus example

We assume the processor and bus technology is the same as described in the example at the end of Chapter 3. For this system with a single memory module, we have $t_r = 2.98\mu\text{s}$ and $k_{\text{lin}} = 0.680\text{ns}$. We assume that memory requests are uniformly distributed among the four memory modules. Thus, since each processor issues a request every $2.98\mu\text{s}$, the mean time between requests from a particular processor to a particular memory module is $2.98\mu\text{s} \times 4 = 11.9\mu\text{s}$. From this, we get $r_{\text{lin}} = 0.680\text{ns}/11.9\mu\text{s} = 0.000057$. The Markov chain model shows that a maximum throughput of 122.8 can be obtained using 134 processors. With a single memory module, a maximum of 67 processors could be used. Thus, using four buses doubles the maximum number of processors. The fact that the maximum number of processors only increases by a factor of two rather than a factor of four using four buses and memory modules can be explained by bus loading; although there are four times as many buses in the 134 processor system, each bus has twice as much load on it and thus runs at only half the speed of the bus in the 67 processor system.

6.2.2 Two-level bus example

We now consider a crosspoint cache system in which a two-level bus hierarchy is used for the memory buses, as shown in Figure 6.1 and Figure 6.2. With the Markov chain model, again using $r_{\text{lin}} = 0.000057$, we find that a maximum throughput of 312.8 can be obtained using 338 processors. In the two-level example of Chapter 3, a maximum throughput of 118.8 was obtained using 136 processors. Thus, an improvement by a factor of 2.49 in the maximum number of processors is obtained by using four buses. From the approximation given in equation (6.1), we would expect an improvement by a factor of $4^{2/3} = 2.52$, which agrees closely with the observed result.

6.3 Summary

We have shown that by using a combination of the two-level bus implementation and the crosspoint cache architecture, it should be feasible to construct very large shared memory multiprocessor systems. An example system design which can support 338 processors was presented.

CHAPTER 7

SUMMARY AND CONCLUSIONS

The single shared bus multiprocessor has been the most commercially successful multiprocessor system design up to this time. Electrical loading problems and limited bandwidth of the shared bus have been the most limiting factors in these systems.

Chapter 3 of this dissertation presents designs for logical buses that will allow snooping cache protocols to be used without the electrical loading problems that result from attaching all processors to a single bus. A new bus bandwidth model was developed that considers the effects of electrical loading of the bus as a function of the number of processors. Using this model, optimal bus configurations can be determined.

In Chapter 4, trace driven simulations show that the performance estimates obtained from the bus model developed in Chapter 3 agree closely with the performance that can be expected when running a realistic multiprogramming workload. The model was also tried with a parallel workload to investigate the effects of violating the independence assumption. It was found that violation of the independence assumption produced large errors in the mean service time estimate, but the bus utilization estimate was still reasonably accurate (within 6% for the workload used).

In Chapter 5, a new system organization was proposed to allow systems with more than one memory bus to be constructed. This architecture is essentially a crossbar network with a cache memory at each crosspoint. A two-level cache organization is appropriate for this architecture. A small cache may be placed close to each processor, preferably on the CPU chip, to minimize the effective memory access time. A larger cache built from slower, less expensive memory is then placed at each crosspoint to minimize the bus traffic.

In Chapter 6, it was shown that by using a combination of the logical bus implementations described in Chapter 3 and the crosspoint cache architecture presented in Chapter 5, it should be feasible to construct shared memory multiprocessor systems with several hundred processors.

7.1 Future research

In Chapter 3, only the two-level bus hierarchy and the binary tree interconnection were studied in detail. It would be useful to find a method for determining the optimal interconnection topology as a function of the number of processors and the delay characteristics of the transceiver technology.

The range of workloads considered in Chapter 4 could be expanded. To provide input data for future simulation studies, a much larger collection of realistic programs could be used, both for multiprogramming and parallel algorithm workloads. Also, performance could be investigated in a mixed environment in which both multiprogramming and parallel workloads coexist.

In Chapter 5, a simple protocol to enforce cache consistency with direct mapped caches was presented. Further studies could investigate protocols suitable for use with set associative caches. Also, alternative protocols could be investigated. For example, the protocol proposed in Chapter 5 used a write through policy for the on-chip caches. This ensured that the crosspoint caches always had the most current data so that snoop hits could be serviced quickly. An alternative approach would be to use a copy back policy for the on-chip caches, which would reduce the traffic on the processor buses but would significantly complicate the actions required when a snoop hit occurs. Performance studies of these alternative implementations could be performed.

Another interesting topic for future work would be to investigate the most efficient ways of incorporating virtual memory support into the crosspoint cache architecture. Tradeoffs between virtual and physical address caches and could be investigated, along with tradeoffs involving translation lookaside buffer placement.

In summary, this work may be extended by considering alternative bus implementations, more diverse workloads, and additional cache implementations and policies, and by investigating the implications of supporting virtual memory.

REFERENCES

REFERENCES

[AB84]
James Archibald and Jean-Loup Baer. "An Economical Solution to the Cache Coherence Problem". *The 11th Annual International Symposium on Computer Architecture Conference Proceedings*, Ann Arbor, Michigan, IEEE Computer Society Press, June 5–7, 1984, pages 355–362.

[AB86]
James Archibald and Jean-Loup Baer. "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model". *ACM Transactions on Computer Systems*, volume 4, number 4, November 1986, pages 273–298.

[AM87]
Russell R. Atkinson and Edward M. McCreight. "The Dragon Processor". *Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, California, IEEE Computer Society Press, October 5–8, 1987, pages 65–69.

[Archi88]
James K. Archibald. "A Cache Coherence Approach For Large Multiprocessor Systems". *1988 International Conference on Supercomputing*, St. Malo, France, ACM Press, July 4–8, 1988, pages 337–345.

[Balak87]
Balu Balakrishnan. "32-Bit System Buses — A Physical Layer Comparison". *Computing*, August 27, 1987, pages 18–19 and September 10, 1987, pages 32–33.

[Bell85]
C. Gordon Bell. "Multis: A New Class of Multiprocessor Computers". *Science*, volume 228, number 4698, April 26, 1985, pages 462–467.

[Bhand75]
Dileep P. Bhandarkar. "Analysis of Memory Interference in Multiprocessors". *IEEE Transactions on Computers*, volume C-24, number 9, September 1975, pages 897–908.

[Bhuya84]
Laxmi N. Bhuyan. "A combinatorial analysis of multibus multiprocessors". *Proceedings of the 1984 International Conference on Parallel Processing*, IEEE Computer Society Press, August 21–24, 1984, pages 225–227.

[BKT87]

BOB BECK, BOB KASTEN, AND SHREEKANT THAKKAR. "VLSI Assist For A Multiprocessor". *Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, California, IEEE Computer Society Press, October 5–8, 1987, pages 10–20.

[Borri85]

PAUL L. BORRILL. "MicroStandards Special Feature: A Comparison of 32-Bit Buses". *IEEE Micro*, IEEE Computer Society, volume 5, number 6, December 1985, pages 71–79.

[Bowlb85]

REED BOWLBY. "The DIP may take its final bows". *IEEE Spectrum*, volume 22, number 6, June 1985, pages 37–42.

[BS76]

FOREST BASKETT AND ALAN JAY SMITH. "Interference in Multiprocessor Computer Systems with Interleaved Memory". *Communications of the ACM*, volume 19, number 6, June 1976, pages 327–334.

[BW87]

JEAN-LOUP BAER AND WEN-HANN WANG. "Architectural Choices for Multilevel Cache Hierarchies". *Proceedings of the 1987 International Conference on Parallel Processing*, The Pennsylvania State University Press, August 17–21, 1987, pages 258–261.

[BW88]

JEAN-LOUP BAER AND WEN-HANN WANG. "On the Inclusion Properties for Multi-Level Cache Hierarchies". *The 15th Annual International Symposium on Computer Architecture Conference Proceedings*, Honolulu, Hawaii, IEEE Computer Society Press, May 30–June 2, 1988, pages 73–80.

[EA85]

KHALED A. EL-AYAT AND RAKESH K. AGARWAL. "The Intel 80386 — Architecture and Implementation". *IEEE Micro*, IEEE Computer Society, volume 5, number 6, December 1985, pages 4–22.

[EK88]

SUSAN J. EGGERS AND RANDY H. KATZ. "A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation". *The 15th Annual International Symposium on Computer Architecture Conference Proceedings*, Honolulu, Hawaii, IEEE Computer Society Press, May 30–June 2, 1988, pages 373–382.

[Encor85]

Multimax Technical Summary. Encore Computer Corporation, May 1985.

[GA84]
Ambuj Goyal and Tilak Agerwala. "Performance Analysis of Future Shared Storage Systems". *IBM Journal of Research and Development*, volume 28, number 1, January 1984, pages 95–108.

[GC84]
James R. Goodman and Men-chow Chiang. "The Use of Static Column RAM as a Memory Hierarchy". *The 11th Annual International Symposium on Computer Architecture Conference Proceedings*, Ann Arbor, Michigan, IEEE Computer Society Press, June 5–7, 1984, pages 167–174.

[Goodm83]
James R. Goodman. "Using Cache Memory to Reduce Processor–Memory Traffic". *The 10th Annual International Symposium on Computer Architecture Conference Proceedings*, Stockholm, Sweden, IEEE Computer Society Press, June 13–17, 1983, pages 124–131.

[HELT*86]
Mark Hill, Susan Eggers, Jim Larus, George Taylor, Glenn Adams, B. K. Bose, Garth Gibson, Paul Hansen, Jon Keller, Shing Kong, Corinna Lee, Daebum Lee, Joan Pendleton, Scott Ritchie, David Wood, Ben Zorn, Paul Hilfinger, Dave Hodges, Randy Katz, John Ousterhout, and Dave Patterson. "Design Decisions in SPUR". *Computer*, volume 19, number 10, November 1986, pages 8–22.

[Hill87]
Mark Donald Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. Ph.D. dissertation, Report Number UCB/CSD 87/381, Computer Science Division, Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, November 25, 1987.

[Hill88]
Mark D. Hill. "A Case for Direct-Mapped Caches". *Computer*, volume 21, number 12, December 1988, pages 25–40.

[Hooge77]
Cornelis H. Hoogendoorn. "A General Model for Memory Interference in Multiprocessors". *IEEE Transactions on Computers*, volume C-26, number 10, October 1977, pages 998–1005.

[HS84]
Mark D. Hill and Alan Jay Smith. "Experimental Evaluation of On-Chip Microprocessor Cache Memories". *The 11th Annual International Symposium on Computer Architecture Conference Proceedings*, Ann Arbor, Michigan, IEEE Computer Society Press, June 5–7, 1984, pages 158–166.

[Humou85]

Humoud B. Humoud. *A Study in Memory Interference Models.* Ph.D. dissertation, The University of Michigan Computing Research Laboratory, Ann Arbor, Michigan, April 1985.

[KEWPS85]

R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. "Implementing a Cache Consistency Protocol". *The 12th Annual International Symposium on Computer Architecture Conference Proceedings*, Boston, Massachusetts, IEEE Computer Society Press, June 1985, pages 276–283.

[LV82]

Tomás Lang and Mateo Valero. "M-Users B-Servers Arbiter for Multiple-Busses Multiprocessors". *Microprocessing and Microprogramming 10*, North-Holland Publishing Company, 1982, pages 11–18.

[LVA82]

Tomás Lang, Mateo Valero, and Ignacio Alegre. "Bandwidth of Crossbar and Multiple-Bus Connections for Multiprocessors". *IEEE Transactions on Computers*, volume C-31, number 12, December 1982, pages 1227–1234.

[MA84]

Trevor N. Mudge and Humoud B. Al-Sadoun. "Memory Interference Models with Variable Connection Time". *IEEE Transactions on Computers*, volume C-33, number 11, November 1984, pages 1033–1038.

[MA85]

Trevor N. Mudge and Humoud B. Al-Sadoun. "A semi-Markov model for the performance of multiple-bus systems". *IEEE Transactions on Computers*, volume C-34, number 10, October 1985, pages 934–942.

[MC80]

Carver Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980, pages 12–14.

[MHBW84]

Trevor N. Mudge, John P. Hayes, Gregory D. Buzzard, and Donald C. Winsor. "Analysis of Multiple Bus Interconnection Networks". *Proceedings of the 1984 International Conference on Parallel Processing*, IEEE Computer Society Press, August 21–24, 1984, pages 228–235.

[MHBW86]

Trevor N. Mudge, John P. Hayes, Gregory D. Buzzard, and Donald C. Winsor. "Analysis of multiple-bus interconnection networks". *Journal of Parallel and Distributed Computing*, volume 3, number 3, September 1986, pages 328–343.

[MHW87]
TREVOR N. MUDGE, JOHN P. HAYES, AND DONALD C. WINSOR. "Multiple Bus Architectures". *Computer*, volume 20, number 6, June 1987, pages 42–48.

[Motor83]
Motorola Schottky TTL Data. Motorola Inc., 1983.

[Motor84]
MC68020 32-Bit Microprocessor User's Manual. Motorola, Inc., Prentice Hall, Inc., 1984.

[Motor88]
Motorola Memory Data. Motorola Inc., 1988.

[MR85]
DOUG MACGREGOR AND JON RUBINSTEIN. "A Performance Analysis of MC68020-based Systems". *IEEE Micro*, IEEE Computer Society, volume 5, number 6, December 1985, pages 50–70.

[PFL75]
R. C. PEARCE, J. A. FIELD, AND W. D. LITTLE. "Asynchronous arbiter module". *IEEE Transactions on Computers*, volume C-24, number 9, September 1975, pages 931–932.

[PGHLNSV83]
DAVID A. PATTERSON, PHIL GARRISON, MARK HILL, DIMITRIS LIOUPIS, CHRIS NYBERG, TIM SIPPEL, AND KORBIN VAN DYKE. "Architecture of a VLSI Instruction Cache for a RISC". *The 10th Annual International Symposium on Computer Architecture Conference Proceedings*, Stockholm, Sweden, IEEE Computer Society Press, June 13–17, 1983, pages 108–116.

[Phill85]
DAVID PHILLIPS. "The Z80000 Microprocessor". *IEEE Micro*, IEEE Computer Society, volume 5, number 6, December 1985, pages 23–36.

[PP84]
MARK S. PAPAMARCOS AND JANAK H. PATEL. "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories". *The 11th Annual International Symposium on Computer Architecture Conference Proceedings*, Ann Arbor, Michigan, IEEE Computer Society Press, June 5–7, 1984, pages 348–354.

[RJ87]
ABHIRAM G. RANADE AND S. LENNART JOHNSSON. "The Communication Efficiency of Meshes, Boolean Cubes and Cube Connected Cycles for Wafer Scale Integration". *Proceedings of the 1987 International Conference on Parallel Processing*, The Pennsylvania State University Press, August 17–21, 1987, pages 479–482.

[RS84]

LARRY RUDOLPH AND ZARY SEGALL. "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors". *The 11th Annual International Symposium on Computer Architecture Conference Proceedings*, Ann Arbor, Michigan, IEEE Computer Society Press, June 5–7, 1984, pages 340–347.

[Seque86]

Balance Technical Summary. Sequent Computer Systems, Inc., November 19, 1986.

[SG83]

JAMES E. SMITH AND JAMES R. GOODMAN. "A Study of Instruction Cache Organizations and Replacement Policies". *The 10th Annual International Symposium on Computer Architecture Conference Proceedings*, Stockholm, Sweden, IEEE Computer Society Press, June 13–17, 1983, pages 132–137.

[SL88]

ROBERT T. SHORT AND HENRY M. LEVY. "A Simulation Study of Two-Level Caches". *The 15th Annual International Symposium on Computer Architecture Conference Proceedings*, Honolulu, Hawaii, IEEE Computer Society Press, May 30–June 2, 1988, pages 81–88.

[Smith82]

ALAN JAY SMITH. "Cache Memories". *Computing Surveys*, Association for Computing Machinery, volume 14, number 3, September 1982, pages 473–530.

[Smith85a]

ALAN JAY SMITH. "Cache Evaluation and the Impact of Workload Choice". *The 12th Annual International Symposium on Computer Architecture Conference Proceedings*, Boston, Massachusetts, IEEE Computer Society Press, June 1985, pages 64–73.

[Smith85b]

ALAN JAY SMITH. 'CPU Cache Consistency with Software Support and Using "One Time Identifiers" '. *Proceedings of the Pacific Computer Communications Symposium*, Seoul, Republic of Korea, October 21–25, 1985, pages 142–150.

[Smith87a]

ALAN JAY SMITH. "Design of CPU Cache Memories". Report Number UCB/CSD 87/357, Computer Science Division, Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, June 19, 1987.

[Smith87b]

ALAN JAY SMITH. "Line (Block) Size Choice for CPU Cache Memories". *IEEE Transactions on Computers*, volume C-36, number 9, September 1987, pages 1063–1075.

[Strec70]
William Daniel Strecker. *An Analysis of the Instruction Execution Rate in Certain Computer Structures.* Ph.D. dissertation, Carnegie-Mellon University, July 28, 1970.

[Towsl86]
Don Towsley. "Approximate Models of Multiple Bus Multiprocessor Systems". *IEEE Transactions on Computers*, volume C-35, number 3, March 1986, pages 220–228.

[TS87]
Charles P. Thacker and Lawrence C. Stewart. "Firefly: a Multiprocessor Workstation". *Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, California, IEEE Computer Society Press, October 5–8, 1987, pages 164–172.

[VLZ88]
Mary K. Vernon, Edward D. Lazowska, and John Zahorjan. "An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols". *The 15th Annual International Symposium on Computer Architecture Conference Proceedings*, Honolulu, Hawaii, IEEE Computer Society Press, May 30–June 2, 1988, pages 308–315.

[Wilso87]
Andrew W. Wilson Jr. "Hierarchical Cache / Bus Architecture for Shared Memory Multiprocessors". *The 14th Annual International Symposium on Computer Architecture Conference Proceedings*, Pittsburgh, Pennsylvania, IEEE Computer Society Press, June 2–5, 1987, pages 244–252.

[WM87]
Donald C. Winsor and Trevor N. Mudge. "Crosspoint Cache Architectures". *Proceedings of the 1987 International Conference on Parallel Processing*, The Pennsylvania State University Press, August 17–21, 1987, pages 266–269.

[WM88]
Donald C. Winsor and Trevor N. Mudge. "Analysis of Bus Hierarchies for Multiprocessors". *The 15th Annual International Symposium on Computer Architecture Conference Proceedings*, Honolulu, Hawaii, IEEE Computer Society Press, May 30–June 2, 1988, pages 100–107.