MEMORY SYSTEM DESIGN FOR BUS BASED MULTIPROCESSORS

by

Men-Chow Chiang

A thesis submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN -- MADISON

1991

Chapter 1

Introduction

1.1. Motivation for this Dissertation

Low-cost microprocessors have led to the construction of small- to medium-scale shared memory multiprocessors with a shared bus interconnect. Such multiprocessors, which have been referred to as *multis* by Bell [13], are popular for two reasons: i) the shared bus interconnect is easy to implement and ii) the shared bus interconnect allows an easy solution to the cache coherence problem [29]. Currently, many major computer manufacturers have commercial products that use the multi paradigm.

A typical shared bus, shared memory multiprocessor (hereafter called a multi in this thesis) is shown in Figure 1. The multi consists of several processors (typically microprocessors) connected together to a memory system. The memory system includes the private caches of each processor, the shared bus interconnect, and the main memory. The overall performance of such a multi is heavily influenced by the design of the memory system. Starting with processors at a particular performance level, the multi designer must provide an adequate-performance

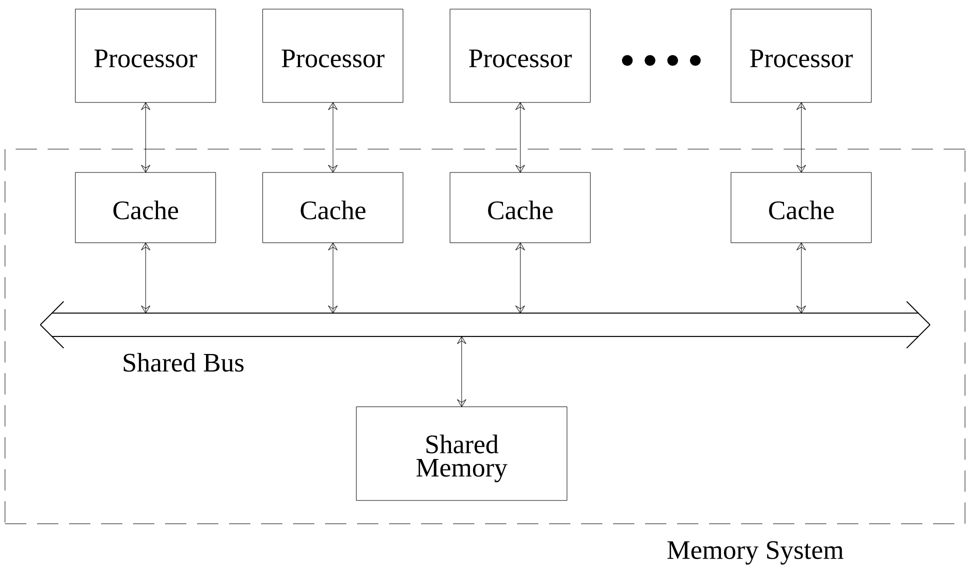


Figure 1.1: A Shared Bus Multiprocessor (Multi)

memory system in order to design a multi with a desired performance level,

To explore the design space of the memory system of a multiprocessor, a suitable methodology needs to be developed to evaluate its performance. A favorite tool of a computer architect is trace-driven simulation using traces generated by the actual execution of sample benchmark programs. Unfortunately, trace-driven simulation may be too time consuming to be practical for evaluating the performance of a memory system. Sometimes the results of simulation may not even be accurate when parallel traces are used (details explained in the following sections). Therefore the first step in this dissertation is to find a more suitable methodology that is both practical and accurate. During the course this study, however, it became clear that the best methodology depends heavily on the intended use, and thus the operating environment of the multiprocessor. Two major operating environments - *throughput-oriented* and *speedup-oriented*, are considered in this thesis.

1.2. Operating Environment of Multiprocessors

1.2.1. Throughput-Oriented Environment

In a *throughput-oriented* environment each processor is executing an independent task, and each independent task is being executed only on one processor. The performance of the system is measured by the overall throughput of the multiprocessor[1]. Therefore the use and the design goal of a multiprocessor operating in a throughput-oriented environment are the same as that of a traditional uniprocessor system, i.e., maximize the throughput of the multiprocessor. The motivation for the multiprocessor is its cost-effectiveness relative to a uniprocessor system.

One major problem faced with designing a throughput-oriented multiprocessor instead of a uniprocessor is the much larger design space to be considered in the multiprocessor. Therefore the first step of study in this environment is to find a performance evaluation method that is both *accurate* and *time-efficient* so that the design space can be more extensively investigated. Trace-driven simulation is certainly an accurate method (if traces used are representative), but simulation is so time consuming that it is not practical for evaluating the whole design space. In this thesis mean value analysis (MVA) analytical models that fulfill the above two requirements are developed and used for the purpose.

---

[1]Another important performance metric in a throughput-oriented environment is the response time of the multiprocessor. Since minimizing the response time and maximizing the overall throughput can not be pursued at the same time, in this dissertation it is assumeed that the *goal* of a multiprocessor operating in the throughput-oriented environment is to maximize the overall system throughput.

However, the accuracy of an analytical model is always a subject of debate unless the accuracy of the model has been established. In this thesis the MVA models are validated by comparing their results with the results of trace-driven simulation. The validated models are then used to explore extensively the design space of a multiprocessor.

1.2.2. Speedup-Oriented Environment

The motivation for using a multiprocessor operating in a *speedup-oriented* environment is to take advantage of parallel processing. Parallel processing has long been recognized as an effective way to deliver much higher performance than uniprocessing, and is usually achieved by running parallel programs on a multiprocessor. Since the interaction between parallel programs and multiprocessors is quite different from that between serial programs and uniprocessor systems, much conventional wisdom in designing uniprocessors may no longer be applicable to multiprocessors. Therefore to find a proper design of multiprocessors requires new study specific to multiprocessors, starting from the fundamental properties of parallel programs and their relationship with multiprocessors.

When a multiprocessor is used to run parallel programs, the most important performance metric is the *speedup*, or the ratio of the execution times between the serial and the parallel solutions of a problem. The major goal of designing a multiprocessor is to achieve the maximum possible speedup for the multiprocessor. Ideally the more processors we use, the higher speedup we can achieve, as long as there is sufficient parallelism in the programs. But the inherent parallelism available in a problem is often not fully utilized by multiprocessors. Therefore the first objective of my study in speedup-oriented environment is to find out how different hardware and software factors limit the potential speedup of parallel programs, and how they impact the memory system.

The execution characteristics of some programs will be examined by using different parallelizing strategies running under different multiprocessor configurations. With the detailed knowledge of the program behavior, the next stage of my research is to suggest different ways to improve the memory system performance of a multiprocessor. The magnitude of improvement is also evaluated in the thesis.

1.3. Previous Work

Since numerous research issues are associated with the design of multiprocessors, previous works on the topic have been very diverse. As these earlier works, this dissertation can only concentrate on certain aspects of multiprocessor designs. Therefore those works relevant to this dissertation will be described only in the chapters themselves.

1.4. Organization of the Dissertation

Since the methodologies used for the two operating environments of multiprocessors are quite different from each other, the content of this thesis is conveniently divided into two parts. The first part studies a throughput-oriented environment, and includes Chapter 2, 3, and 4. Chapter 2 describes the memory system model and discusses some of the design choices that are considered for evaluation. Chapter 3 introduces the methodology for performance study. At the heart of the methodology are the mean value analysis (MVA) analytical models. The detail of how the models are developed and validated are given. Chapter 4 presents the results of evaluation for some key design choices in the memory system.

The second part of this thesis deals with the speedup-oriented environment. Chapter 5 describes the design issues and the methodology for performance evaluation in this environment. One important point in the methodology is that performance statistics need to be extracted separately for the different computational phases of a running parallel program. Chapter 6 gives a qualitative analysis of the characteristics in different computational phases of a parallel program. Chapter 7 introduces another important part of the evaluation methodology - execution driven simulation. The multiprocessor simulator, parallel benchmarks, and simulation conditions are described in detail. Chapter 8 and Chapter 9 present the simulation results, separately for each computational phase. One major finding is that synchronization can become a performance bottleneck even before the bandwidth of the shared bus. Chapter 10 evaluates the performance of several hardware and software synchronization alternatives, including a new synchronization method that uses a *lock table*. Finally Chapter 11 presents the conclusions of this thesis study, and gives suggestions for future work.

1.5. Research Contributions

The major contribution of this dissertation research is that new methodologies have been taken to evaluate the system performance in both operating environments of multiprocessors. In the throughput-oriented environment the Customized Mean Value Analysis (CMVA) models for the memory system of bus-based multiprocessors are developed and the accuracy of the models thoroughly validated by actual trace-driven simulations. The extremely time-efficient models make it possible to explore more extensively the design space of a bus-based multiprocessor.

For the speedup-oriented environment a multiprocessor simulator that takes into account actual timings of system components has been developed to conduct execution-driven simulation. The multiprocessor simulation gives many useful insights to the execution of parallel programs. The detailed and accurate statistics obtained with execution-driven simulation can not be obtained with trace-driven simulation, the best method used in previous research. With this powerful multiprocessor simulator the performance of several synchronization alternatives to the basic *test&set* or *test&test&set* are also evaluated. Among those synchronization methods considered, the *lock table* method, which is first proposed in this thesis, has been found to perform exceedingly well. With its generality and scalability the lock table method may become an important synchronization mechanism in future multiprocessors.

Chapter 2
Throughput Oriented Environment

2.1. Memory System Model

The memory system of a typical multi consists of three main components: i) the private caches of each processor (which may be multi-level), ii) the shared bus interconnect and iii) the shared main memory. In a throughput-oriented environment, the overall throughput of the multiprocessor is simply the sum of the throughput of the individual processors. The throughput of each processor *T* is:

$$T = \frac{1}{IT + M_{ref}\,T_m^P} \tag{2.1}$$

where

- IT is the average instruction execution time, assuming all memory accesses take no time.
- M_{ref} is the average number of memory references generated by an instruction.
- T_m^P is the average memory access time (or latency) seen by a processor.

The interaction between the processors themselves, and the memory system is reflected through T_m^P. Maximizing the throughput therefore implies minimizing T_m^P. T_m^P can be further developed as:

$$T_m^P = T_C^P + M \times T_m^C \tag{2.2}$$

where

- T_C^P is the cache access time.
- M is the cache miss ratio.
- T_m^C is the average time taken to service a cache miss[1].

Equation (2.2) is applicable in general to any processing system (uniprocessor or multiprocessor) with a cache and a memory. However, the components of the equation are variable and depend upon the parameters of the memory system. For example, T_C^P is a function only of the cache organization. Likewise, M is also a function only

[1] T_m^C is also the *effective* cache miss latency seen by a processor, i.e., only one outstanding memory request is allowed for each processor, and for the duration of the memory access the processor is blocked from executing any instruction. This is true in shared bus multiprocessors that use microprocessors as their CPUs. Most microprocessors allow only a single outstanding memory request.

of the cache organization and is not dependent on the other parameters of the memory system. However, T_m^C can be a function of several parameters of the memory system, such as the characteristics of the shared bus and the main memory.

A major difference between the design of a memory system for a uniprocessor and a multiprocessor operating in a throughput-oriented environment is in the impact of T_m^C. In a uniprocessor, T_m^C can be approximated as $T_m^C = \alpha + \beta \cdot B$, where B is the cache block size and α and β are constants that represent the fixed overhead and the unit transfer cost of transferring a cache block [62].

In a multi, T_m^C cannot be approximated simply as $T_m^C = \alpha + \beta \cdot B$. This is because T_m^C includes a queuing delay that can have a significant overall contribution to T_m^P. This queuing delay is dependent upon the utilization of the bus, which in turn is dependent upon several system-wide characteristics such as: i) the traffic on the bus, which is influenced by the number of processors connected to the bus, and the organizations of their caches, ii) the bus switching strategy, and iii) the main memory latency. If accurate results are to be obtained for design choices in shared bus multiprocessor memory systems, all system factors and their complex interdependencies must be taken into account.

Before proceeding further, let us consider some design choices in the three main components of the memory system of a multi and see how they might influence one another. A comprehensive evaluation of design choices is not possible in this thesis (but can be carried out with the methodology here) and the attention will be focused on some key design choices for throughput oriented multiprocessors.

2.2. Design Choices

2.2.1. Cache Memory

The key component of the memory system is the cache and most of the issues are concerned with how the choice of cache parameters influences the design of the other components and vice-versa. Three important cache parameters are considered: i) the cache size, ii) the cache block size and iii) the cache set associativity.

A large cache is able to lower T_m^P directly by lowering M.[2] Furthermore, because of lower bus traffic

[2]When a multi is executing parallel programs, M can vary even for a fixed size cache. The value of M depends on the number of processors on which the parallel program is executing, as a result of a phenomenon called *reference spreading* [44]. Consequently, a larger cache may not necessarily be able to lower the value of M. However, for a throughput-oriented multi, there is no reference spreading and a larger cache will result in a lower value of M.

(assuming a constant number of processors N), the utilization of the bus and the queuing delay for a bus request is lowered and consequently T_m^C is reduced. Alternately, a lower per-processor bus utilization allows more processors to be connected together on the bus, possibly increasing the peak throughput (or processing power) of the multi.

While it is clear that larger caches allow more processors to be connected together in a throughput-oriented environment by reducing per-processor bus bandwidth demands and improving T_m^P, the relationship between cache size and other memory system parameters (such as block size, set associativity, bus switching strategy, bus width and main memory latency) needs to be investigated and the improvement quantified.

Cache block size is perhaps the most important parameter in the design of each cache. Block size not only dictates the performance but also the implementation cost of the cache (smaller block sizes result in a larger tag memory than larger block sizes). In a detailed study, Smith mentions several architectural factors that influence the choice of a block size and evaluates them in a uniprocessor environment [62]. Smith's main result of interest to us is that the miss ratio decreases with increasing block size up to a point at which the internal cache interference increases the miss ratio. However, larger block sizes also cause more *traffic* on the cache-memory interconnect. Since this additional traffic uses more bus bandwidth and since the bandwidth of the shared bus is the critical resource in a multi, Goodman has suggested that small block sizes are preferable for cache memories in multis [29].

As Smith points out, minimizing the bus traffic alone is not the correct optimization procedure in multiprocessors and neither is a minimization of the miss ratio, independent of the other parameters of the memory system [62]. This point, which is also apparent from equation (2.2), is central to the performance study of multiprocessors and cannot be overemphasized. If maximizing multiprocessor system throughput is the goal, the choice of the block size should not be decoupled from the parameters of the shared bus and the main memory. Furthermore, any evaluation must consider equation (2.2) in its complete generality and include not only the main memory latency and the bus transfer time of a request, but also the queuing delays experienced by the memory request.

Cache set associativity is also an important design consideration. It is well known that a larger set associativity reduces M (in most cases) and, if T_C^P and T_m^C are constant, a larger set associativity is preferable, subject to implementation constraints [61]. However, as several researchers have observed, T_C^P is not independent of the cache set associativity since a large set associativity requires a more complex implementation and consequently has a higher T_C^P. If the decrease in T_C^P by going to a lower set associativity is greater than the increase in $M \times T_m^C$, then a lower set associativity results in a lower overall T_m^P, and consequently a higher processor throughput.

In a uniprocessor, T_m^C is a constant for a given block size and memory configuration. If M is sufficiently small (because of a large cache size, for example), the decrease in T_C^P due to a lower set associativity can easily overcome an increase in $M \times T_m^C$ [34, 35, 56]. In a multiprocessor, however, T_m^C contains a queuing delay, which can be a large fraction of T_m^C if the bus utilization is high. Increasing cache set associativity not only decreases T_m^P directly by reducing M, it also reduces T_m^P indirectly by reducing the utilization of the bus and consequently the queuing delay component of T_m^C. Therefore, the impact of cache set associativity on multiprocessor memory system design is another important design issue that needs to be investigated.

2.2.2. Shared Bus

The main design issues in the shared bus are the choice of bus width and the bus switching strategy (or protocol). Using a wider bus is an effective way to increase bus bandwidth. Increasing bus bandwidth reduces T_m^C in two ways: directly by reducing the bus transfer time of a block, and indirectly by reducing the bus queuing delay due to the decreased bus tenure of each memory request. However, increasing the bus width may slow down the bus, because the cycle time of the bus may have to be lengthened to tolerate the larger bus time skew when more bus lines are used. Choice of bus width also affects the design of other modules. For example, the bandwidth of cache and main memory should match that of the bus. This implies that the block size of cache and main memory should be made at least as large as the bus width.

Bus switching methods fall into two broad categories: i) *circuit switched* buses and ii) *split transaction, pipelined* buses (hereafter referred to as STP buses in this thesis)[3]. In a circuit switched bus, the bus is held by the bus master until the entire transaction is complete. The time that the bus is held by the master (or the bus tenure) includes the latency of the slave device. Such a switching strategy is used in most existing bus designs. For example, the block read and block write transactions in the IEEE Futurebus employ a circuit switched protocol [16].

In an STP bus, the bus is not held by the master if the slave device is unable to respond to the request immediately. The bus is released by the master and is made available to other bus requesters. When the slave device is ready to respond to a request, it obtains bus mastership and transfers data to the requesting device. An STP bus is used in the Sequent Balance and Symmetry multiprocessors [11, 25], and is also being considered for the IEEE

[3]More precisely, pipelined buses are only a restricted form of split transaction buses. With a pipelined bus the time between sending a request through the bus and receiving the corresponding reply from the bus is a constant. But a split transaction bus does not restrict itself to respond in a constant time. In this thesis an STP bus is meant to be a split transaction bus.

Futurebus+.

2.2.3. Main Memory

The final component of the multiprocessor memory system is the main memory and the parameter of importance is the main memory latency. Many studies choose to ignore this parameter (or assume that it is a constant). As will be seen in the following study, including main memory latency is crucial since it influences other memory system design parameters such as the cache block size, especially with a circuit switched bus.

Chapter 3
Performance Evaluation Methods for a Throughput-Oriented Environment

3.1. Introduction

For evaluating design choices, the favorite tool of a computer architect is trace-driven simulation using traces generated by the actual execution of sample benchmark programs (called an *actual* trace-driven simulation in this thesis). Unfortunately, trace-driven simulation is expensive, both in execution time and trace storage requirements. The storage expense of actual trace-driven simulation can be reduced by parameterized trace driven simulation. In parametrized simulation, artificial traces are generated on the fly by using random number generators, of which the parameter values may be derived from the characteristics of the actual program traces. Parameterized simulation is still computationally expensive and is generally not considered to be as accurate as actual trace-driven simulation. Finally, one can develop an analytical model. Analytical models generally are much cheaper computationally than trace-driven simulation and consequently allow the designer to explore a much larger design space.

Multiprocessors with arbitrary interconnection networks have been the subject of several previous studies [39, 45, 46, 49, 51]. Studies of bus-based multiprocessor design issues have used trace-driven simulation [22], parameterized simulation [8], as well as analytical modeling [69, 70]. For a system as complex as a multi, ideally a system designer would like to use an *accurate* analytical model to explore the design space with a minimal computational requirement.

Both analytical modeling as well as actual trace-driven simulation will be used in this study. The analytical models are based on a ‘‘customized’’ mean value analysis technique that has been proposed in [70] and applied in [36, 43, 71]. Trace-driven simulation is used to study a few thousand cases and, more importantly, build confidence in the analytical models. Once the validity of the analytical models has been established, the models will be used to evaluate the design choices in the next chapter.

3.2. Customized Mean Value Analysis (CMVA)

The CMVA models build on similar models developed to study bus-based multiprocessors [36, 43, 70, 71]. The CMVA method is appealing because it is simple and intuitive. To start we simply follow the path of a cache miss request and sum up the mean waiting times and processing times along the way to form the equations for the mean cache miss response time.

As mentioned earlier, the operational environment considered here for the multi is a throughput-oriented, general multiuser environment where each processor is running a different user program. It is also assumed that the task characteristics for the tasks executing on each processor are always the same. This type of workload can be created, for example, by executing a program repeatedly on the same processor (but different processors execute different programs).

3.2.1. Processor Execution Model

A processor's execution history can be viewed as consisting of two alternating phases, an *execution* phase and a *blocking* phase. During the execution phase the processor executes instructions uninterrupted, with all memory requests satisfied by its local cache. The processor changes to the blocking phase when it makes a blocking bus request. Two kinds of bus requests are distinguished from each other: blocking and non-blocking ones. A processor cannot proceed unless its blocking request (read miss or invalidation) is satisfied; it can proceed without waiting for its non-blocking request (write back of a dirty block) to finish. The relationship between these events is shown in Figure 3.1.

The throughput of a processor during a time period represented by consecutive execution and blocking phases is the number of instructions executed during the two phases divided by the duration of the two phases. Since the processor is blocked during the blocking phase, the throughput can be calculated as the mean number of instructions executed by the processor during an execution phase, divided by the mean total time of the execution and the blocking phases. The mean number of instructions executed in an execution phase, and the mean length of the phase are derived from the trace-driven simulation of a single processor and its cache since these values are not influenced by

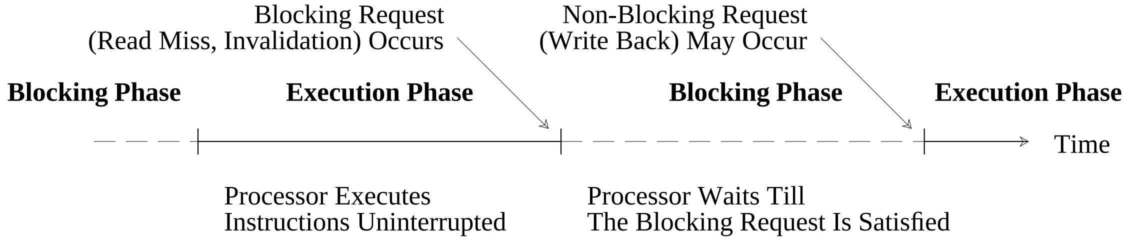


Figure 3.1: Execution History of A Processor

other processors in the system. The mean length of a blocking phase, or the mean response time of a blocking bus request, is calculated using a CMVA model of the shared bus and main memory.

3.2.2. Circuit Switched Bus

The following notation is used:

Input Parameters

- T_e is the mean processing time of a processor between two successive blocking bus requests, i.e., the duration of the execution phase. T_e can be expressed as $IT/(M \times M_{ref})$, where IT is the average instruction execution time, assuming all memory references are cache hits. M is the sum of cache miss and invalidation ratios[1], and M_{ref} is the average number of memory references generated per instruction.
- T_a is the bus arbitration time. One cycle is charged for bus arbitration when a request arrives at the bus and the bus is not busy.
- T_{ro} (T_{vo}, T_{wo}) is the time for which the bus is needed to carry out a read (invalidation, write back) operation, excluding the bus arbitration time. The main memory latency is included as a part of T_{ro} for a circuit switched bus.
- P_r is the probability that a blocking bus request is a read operation.
- P_v is the probability that a blocking bus request is an invalidation operation; note that $P_v + P_r = 1$.
- P_w is the probability that a cache miss results in a write back of a dirty cache block.
- N is the total number of caches (or processors) connected to the shared bus.

Output Parameters

- R is the mean time between two successive blocking bus requests from the same cache.
- Rs_r (Rs_v) is the mean response time of a read (invalidation) request, weighted by P_r (P_v).
- W_{rv} is the mean bus waiting time of a read or an invalidation request.
- T_r (T_v, T_w) is the bus access time of a read (invalidation, write back) request, including the bus arbitration time.
- U_r (U_v, U_w) is the partial utilization of the bus by the reads (invalidations, writes back) from one cache.
- U_{rv} is the partial utilization of the bus by the blocking (read and invalidation) requests from one cache.
- U is the partial utilization of the bus by the requests from one cache; NU is the total bus utilization.
- B_r (B_v, B_w) is the probability that the bus is busy servicing a read (invalidation, write back) request from a particular cache, when a new read or invalidation request arrives.

[1]Invalidation ratio is defined in a similar way to cache miss ratio. A write hit to a clean block generates an invalidation, and invalidation ratio is the percentage of memory references that cause invalidation.

- Re^r (Re^v, Re^w) is the residual service time of a read (invalidation, write back) request, when the request is currently being serviced by the bus when a new read or invalidation request arrives.
- W_w is the mean bus waiting time of a write back request.
- $\overline{Q}_r$ ($\overline{Q}_v$, $\overline{Q}_w$) is the mean number of read (invalidation, write back) requests from the same cache in the bus.
- K^r_{rv} (K^v_{rv}, K^w_{rv}) is the mean waiting time of a read or an invalidation request, due to the read (invalidation, write back) requests already in the bus.
- K^r_w (K^v_w, K^w_w) is the mean waiting time of a write back request, due to the read (invalidation, write back) requests already in the bus.

Response Time Equations

The mean time between two successive blocking bus requests (read miss or invalidation) from the same cache, R, is the sum of T_e and the mean time spent in the blocking phase, which is the weighted mean of the delays of the two types of blocking bus requests. Therefore,

$$R = T_e + Rs_r + Rs_v;\ \text{where}$$

$$Rs_x = P_x\ (W_{rv} + T_x);\quad x = r, v$$

The time that a request spends on the bus is the time that is needed to service the request once it has obtained mastership of the bus, plus any time that might be spent in arbitration for bus mastership. If the bus is busy servicing a request while the arbitration for mastership for the next request takes place, the arbitration time is overlapped completely and does not contribute to the time spent by a request on the bus. On the other hand, the entire time to carry out arbitration is added to the time spent on the bus by a request if the request arrives when the bus is free. The arbitration time component of a request's bus tenure is approximated by considering it to be proportional to the probability that the bus is busy when a request from a cache arrives.

The probability that the bus is idle is $(1 - NU)$. However, since a cache can have only one outstanding blocking request at a time, a blocking request will never see another blocking request from the same cache using the bus when the request reaches the bus. The fraction of time that the bus is servicing a blocking request from a particular cache is U_{rv}. A new blocking request from the same cache can therefore arrive at the bus only during the remaining fraction of time, i.e., $(1 - U_{rv})$. Of this fraction, $(NU - U_{rv})$ is spent servicing other requests. Therefore, the probability that the bus is busy when a blocking request arrives from a cache is $\frac{NU - U_{rv}}{1 - U_{rv}}$, and the probability that the bus is idle is $(1 - \frac{NU - U_{rv}}{1 - U_{rv}})$ [70].

For a non-blocking request (write back) this probability becomes $(1 - \frac{NU - U}{1 - U})$. U instead of U_{rv} is used because it is assumed that a write back request can only be issued immediately after a cache block is returned from the main memory (a result of an earlier blocking request on a cache miss), and it should never see any other request, blocking or non-blocking, from the same cache using the bus. Therefore, the total bus access times for blocking and non-blocking requests are:

$$T_x = (1 - \frac{NU - U_{rv}}{1 - U_{rv}}) \times T_a + T_{xo} = \frac{1 - NU}{1 - U_{rv}} \times T_a + T_{xo}; \quad x = r,\ v$$

$$T_w = (1 - \frac{NU - U}{1 - U}) \times T_a + T_{wo} = \frac{1 - NU}{1 - U} \times T_a + T_{wo}$$

Waiting Time Equations

Using the mean value technique for queuing network models [40], The waiting time of an arriving request is decomposed into three components based on the types of the requests that delay the service of the new request. For a blocking request:

$$W_{rv} = K_{rv}^{r} + K_{rv}^{v} + K_{rv}^{w}$$

where

$$K_{rv}^{x} = (N - 1)\left((\overline{Q}_x - B_x) \times T_x + B_x \times \mathrm{Re}^{x}\right); \quad x = r,\ v$$

$$K_{rv}^{w} = N\left((\overline{Q}_w - B_w) \times T_w + B_w \times \mathrm{Re}^{w}\right)$$

The residual service time for the request that is being serviced when a new read or invalidation request arrives is [40]:

$$\mathrm{Re}^{x} = \frac{T_x}{2}; \quad x = r,\ v,\ w$$

The probabilities that the bus is busy servicing the request from a particular cache when a new read or invalidation request arrives can be approximated as:

$$B_x = \frac{U_x}{1 - U_{rv}}; \quad x = r,\ v,\ w$$

where

$$U_x = \frac{P_x\ T_x}{R}; \quad x = r,\ v$$

$$U_w = \frac{P_w\ P_r\ T_w}{R}$$

$$U_{rv} = U_r + U_v$$

$$U = U_r + U_v + U_w$$

A scaling factor of $(1 - U_{rv})$ is used because when a blocking request such as a read or an invalidation arrives at the bus, it will not see any blocking request from the same cache being serviced by the bus.

The mean number of requests from a particular cache, or the mean partial queue lengths contributed by a particular cache seen by the arriving request can be approximated by:

$$\overline{Q}_x = \frac{Rs_x}{R} = \frac{P_x\ (W_{rv} + T_x)}{R}; \quad x = r,\ v$$

$$\overline{Q}_w = \frac{P_w\ P_r\ (W_w + T_w)}{R}$$

Here the queue lengths include the request that is currently being serviced by the bus. K^r_{rv}, K^v_{rv} and K^w_{rv} can now be computed as:

$$K^x_{rv} = (N-1)\left[\left(\frac{P_x\ (W_{rv} + T_x)}{R} - B_x \right) \times T_x + B_x \times \mathrm{Re}^x \right]; \quad x = r,\ v$$

$$K^w_{rv} = N\left[\left(\frac{P_r\ P_w\ (W_w + T_w)}{R} - B_w \right) \times T_w + B_w \times \mathrm{Re}^w \right]$$

Similarly, the waiting time equations for a write back request can be derived:

$$W_w = K^r_w + K^v_w + K^w_w$$

where

$$K^x_w = (N-1)\,\overline{Q}_x\,T_x = (N-1) \times \frac{P_x\ (W_{rv} + T_x)}{R} \times T_x; \quad x = r,\ v$$

$$K^w_w = (N-1)\,\overline{Q}_w\,T_w = (N-1) \times \frac{P_w\ P_r\ (W_w + T_w)}{R} \times T_w$$

In the above equations for K^x_w and K^w_w, it is assumed that a write back request immediately follows a cache miss read and is issued after the main memory reply to the miss read arrives at the cache. Therefore when the write back request arrives at the bus, the residual service time of the request is simply the complete service time of the request.

3.2.3. STP Bus

The derivation of the CMVA model for an STP bus is carried out along similar lines as in the case of a circuit switched bus. The following additional notations are used for an STP bus:

Input Parameters

- T_m is the main memory latency.
- T_{qo} is the bus access time of a read request, excluding the bus arbitration time.
- T_{do} is the bus access time of a block transfer either for a cache write back or a main memory reply, excluding the bus arbitration time.

Output Parameters

- W_{qv} is the mean bus waiting time of a read or an invalidation request.
- W_d is the mean bus waiting time of a main memory reply.
- T_q is the bus access time of a read operation, including the bus arbitration time.
- T_d is the bus access time of a write back or a main memory reply, including the bus arbitration time.
- U_q denotes the partial utilization of the bus by the read requests from one cache.
- U_d denotes the partial utilization of the bus by the main memory replies to one cache.
- U_r denotes the partial utilization of the bus by the reads, invalidations from, and the main memory replies to one cache.
- $\overline{Q}_q$ denotes the mean number of read requests from the same cache in the bus.
- $\overline{Q}_d$ denotes the mean number of main memory replies to the same cache in the bus.
- B_q (B_v, B_w) is the probability that the bus is busy servicing a read (invalidation, write back) request from a particular cache, when a new read, invalidation, or main memory reply arrives.
- B_d is the probability that the bus is busy servicing a main memory reply to a particular cache, when a new read, invalidation, or main memory reply arrives.
- Re^q is the residual transfer time of a read request when the request is serviced by the bus and a new read, invalidation, or main memory reply arrives.
- Re^d is the residual transfer time of a main memory reply when the memory reply is serviced by the bus and a new read, invalidation, or main memory reply arrives.
- K_r^q (K_r^v, K_r^d) is the mean bus waiting time of a read request, an invalidation from, or a main memory reply, due to the read (invalidation, main memory reply) requests already in the bus.
- K_{qv}^w is the mean bus waiting time of a read request or an invalidation, due to the writes back already in the bus.
- K_d^w is the mean bus waiting time for a main memory reply, due to the writes back already in the bus.

- K_w^d is the mean bus waiting time of a write back request, due to the main memory replies already in the bus.

Response Time Equations

The response time equations of the CMVA model for an STP bus can be derived in a way similar to that for a circuit switched bus.

$$R = T_e + Rs_r + Rs_v$$

$$Rs_r = P_r\ (W_{qv} + T_q + T_m + W_d + T_d)$$

$$Rs_v = P_v\ (W_{qv} + T_v)$$

where

$$T_x = (1 - \frac{NU - U_r}{1 - U_r}) \times T_a + T_{xo} = \frac{1 - NU}{1 - U_r} \times T_a + T_{xo}; \quad x = q,\ v,\ d$$

$$T_w = (1 - \frac{(N-1)\,U}{1 - U}) \times T_a + T_{do} = \frac{1 - NU}{1 - U} \times T_a + T_{do}$$

Waiting Time Equations

The waiting time equations for an STP bus are more complicated than those for a circuit switched bus because there are four kinds of requests in the system: a cache can generate read, write and invalidation requests, and the main memory can generate replies in response to read requests. An arriving request can see all four kinds of requests in the bus queue, hence its average waiting time consists of four components.

$$W_{qv} = K_r^q + K_r^v + K_r^d + K_{qv}^w$$

$$K_r^x = (N-1)\left[(\overline{Q}_x - B_x) \times T_x + B_x \times \mathrm{Re}^x\right]; \quad x = q,\ v,\ d$$

$$K_{qv}^w = N\left[(\overline{Q}_w - B_w) \times T_w + B_w \times \mathrm{Re}^w\right]$$

$$W_d = K_r^q + K_r^v + K_r^d + K_d^w$$

$$K_d^w = (N-1)\left[(\overline{Q}_w - B_w) \times T_w + B_w \times \mathrm{Re}^w\right]$$

$$W_w = K_w^q + K_w^d + K_w^v + K_w^w$$

$$K_w^x = (N-1)\,\overline{Q}_x\, T_x; \quad x = q,\ d,\ v,\ w$$

The multiplication factor for K_{qv}^{w} is N and for K_{d}^{w} is $(N-1)$ because an arriving read or invalidation request may see a write back request from the same cache in the bus, whereas a main memory reply destined for a particular cache will never see a write back from the same cache on the bus. The equations for the residual service time of the request that is currently being serviced, when a new read or invalidation request from some cache, or a reply from main memory arrives, are

$$\mathrm{Re}^{x} = \frac{T_x}{2}; \quad x = q,\ v,\ d,\ w$$

The remaining equations are

$$B_x = \frac{U_x}{1 - U_r}; \quad x = q,\ v,\ d,\ w$$

$$\overline{Q}_q = \frac{P_r\ (W_{qv} + T_q)}{R}$$

$$\overline{Q}_v = \frac{P_v\ (W_{qv} + T_v)}{R}$$

$$\overline{Q}_d = \frac{P_r\ (W_d + T_d)}{R}$$

$$\overline{Q}_w = \frac{P_r\ P_w\ (W_w + T_w)}{R}$$

$$U_x = \frac{P_r\ T_x}{R}; \quad x = q,\ d$$

$$U_v = \frac{P_v\ T_v}{R}$$

$$U_w = \frac{P_r\ P_w\ T_w}{R}$$

$$U_r = U_q + U_d + U_v = \frac{P_r\ (T_q + T_d) + P_v\ T_v}{R}$$

$$U = U_q + U_d + U_v + U_w = \frac{P_r\ (T_q + T_d) + P_v\ T_v + P_r\ P_w\ T_w}{R}$$

3.3. Trace Driven Simulation

3.3.1. Simulators

Trace-driven simulation is carried out using a software simulator that simulates program execution on a Sequent Symmetry-like multiprocessor. The simulator consists of three modules: (i) a program interpreter or tracer, (ii) a cache simulator and (iii) a shared bus (and main memory) simulator.

The benchmark program whose execution is to be simulated is compiled into the Intel 80386 machine language using the Sequent Symmetry C compiler. The tracer program then interprets the program and uses the *ptrace* facility of Dynix to obtain a dynamic *memory trace*. Each memory trace record contains the *virtual memory address* accessed, the *access type* (a read or a write) and the *time* the access is made. The time associated with each memory reference in the trace generated by the tracer program is an ideal number that would represent the time at which the memory reference would be generated if: (i) all memory references generated by an instruction are generated simultaneously and (ii) all memory references are serviced in zero time. To obtain realistic times at which the memory references would be generated and serviced in the multiprocessor environment, the memory traces have to be passed through the cache and bus simulators.

The memory trace generated by the tracer program is used to drive a *cache simulator*. By filtering out references that are cache hits (of course depending upon the cache organization), the cache simulator generates a *cache miss*, *write back*, and *invalidation*[2] trace; *i.e.*, a trace of *bus requests*. Each bus trace record stores the time of generation (still an ideal time) and the type of the bus request. The Berkeley Ownership protocol is used for generating the invalidation requests [38], though any other protocol could be used in a straightforward manner.

Bus traces from several benchmark programs are then used to drive a *bus simulator* which simulates the operation of the shared bus and the main memory. In deciding which request is to be serviced next, the bus simulator uses a FCFS policy. The relevant delays and timing parameters in the simulation model are shown in Figure 3.2. For each input bus request, the latency seen by the request is the sum of bus queuing delay, the bus transfer time of the request and, for a cache miss read, the reply.

As a result of the bus simulation, realistic times at which each memory reference is serviced are obtained. Note that the decoupling of cache and bus simulations is possible due to the assumption of a throughput-oriented multi-user environment. In such an environment the actual memory performance will not affect the order of the events that happen on each processor and cache (this may not be true if the multiprocessor is executing a parallel program). The bus simulation simply calculates a total ordering with a correct time scale for all the events in the system.

---

[2]An invalidation request is generated on a write hit to a clean block, even though this is not necessary in the environment where only non-parallel programs are running, and there is no task migration.

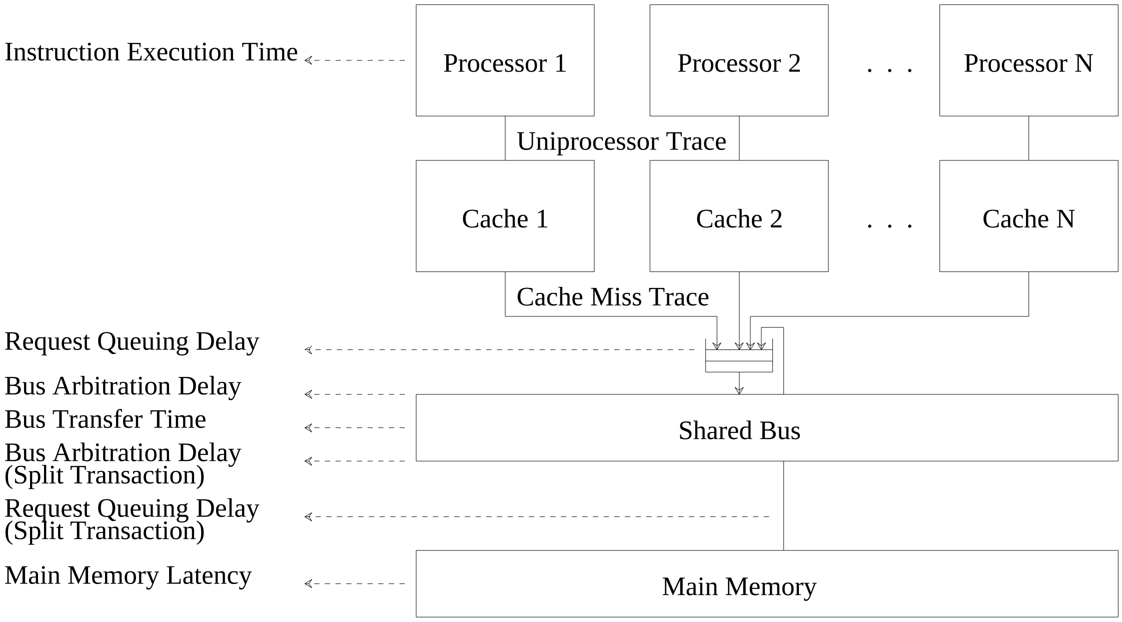


Figure 3.2: Timing Delays in the Trace-Driven Simulation Model

In simulation it is assumed that a processor stalls until its blocking request (cache miss read or invalidation) is serviced, *i.e.*, it can have only one outstanding memory request. It is also assumed that there is no task migration. The latter assumption is made to keep the simulator manageable and does not affect the purpose of the simulator.

3.3.2. Benchmarks

The CMVA models are validated using two kinds of workload: *non-homogeneous* and *homogeneous*.

3.3.2.1. Non-Homogeneous Workload

Two sets of *non-homogeneous* workload are used to validate the CMVA models. The first set consists of several benchmark programs running on Symmetry, or Intel 80386 (I386) processors. The benchmarks are, (i) *as*, which is the assembler for the Intel 80386 processor, (ii) *awk*, the pattern scanning and processing program, (iii) *cache*, the cache simulator itself, (iv) *ccom*, which is a C compiler, (v) *compress*, which compresses a file, (vi) *csh*, the command interpreter, (vii) *nroff*, the nroff text processing program, and (viii) *sort*, which sorts a file lexicographically. For each benchmark, a memory trace is collected for 1 million instructions executed. During this period 1.3 to 1.6 million memory accesses are made by these programs.

The second set is the TITAN traces [15]. The traces are generated from Titan processors, which have a 32-bit, load/store (‘‘RISC’’) architecture designed by DEC. The benchmarks are, (i) *mult*, a multiprogrammed workload, which includes a series of compiles, a printed circuit board router, a VLSI design rule checker, and a series of simple UNIX commands, all executing in parallel (approximately 40 megabytes active at any time) with an average of 214,000 instructions executed between each process switch., (ii) *tv*, a VLSI timing verifier (96 megabytes), (iii) *sor*, a uniprocessor successive over-relaxation algorithm which uses very large, sparse matrices (62 megabytes), and (iv) *tree*, a Scheme program which searches a large tree data structure (64 megabytes). Note that data sizes in these benchmarks are very large, because these memory traces are originally used to evaluate very large caches.

For each Titan benchmark two memory traces are generated during two different sampling periods of execution. Each trace includes about 1.3 to 1.6 million memory accesses for the duration of 1 million instructions. Therefore the TITAN workload also contains eight traces. It should be noted that these memory traces are generated from Titan processors, not obtained from running the tracer module of the multiprocessor simulator with these benchmarks on Symmetry. Therefore this set of traces reflect the architecture of Titan processors.

3.3.2.2. Homogeneous Workload

A set of homogeneous workload is derived from four of the above I386 memory traces: *as, cache, csh, and nroff*. The homogeneous workload is created by concatenating the four traces for each processor. Different processor receives a concatenation of the traces in different order. In this way bursts of bus references from processors are less likely to coincide with each other, and at the same time each processor runs a workload with the same overall characteristics as in any other processor.

3.3.3. Simulation Method and Conditions

This section describes the way to conduct simulation with non-homogeneous workload. The case with homogeneous workload is quite similar (the only difference is the trace used for each processor), and will not be discussed.

For each set of workload separate single processor cache simulations are conducted using the eight memory traces. For each cache configuration, and for each memory trace, a set of statistics is collected and a bus request trace file generated. The statistics are used as inputs to the CMVA models, and the bus request trace file is used to drive the bus simulator. Cache statistics and the bus request trace are generated for the entire length of a memory trace and include the cold start effect of the cache. Cold start is assumed because when large caches are used in a multiprocessor, the initial cold-start load is a major part of the bus activity. All caches are copy back (or write back), and use LRU replacement.

A set of eight cache miss traces generated with the same cache configuration are used to drive the bus simulator in a simulation. Since each traced program is assumed to run on one processor and never migrate, the term *process, traced program*, or *processor* can be used interchangeably. Two multiprocessor configurations are considered, one with 8 processors each running a separate benchmark program, and the other with 16 processors with each benchmark program being run simultaneously by 2 processors. This represents a **non-homogeneous** environment in which each processor is running a task with a different characteristic.

To allow for a comparison between the models and simulation, the following two conditions should be met: (i) simulation is carried out for a sufficiently long time so that the transients have subsided[1], and (ii) both model and simulation correspond to the same scenario.

The length of simulation is measured by the number of bus cycles (not by how many processor instructions) that have been simulated. Computing resource constraints prevent us from simulating for more than a few hundred million bus cycles for each simulation run. For small caches that generate a lot of bus activity, several hundred million bus cycles for each processor could be simulated with only a few (2 or 3) passes through a trace. For larger caches that generate smaller amounts of bus activity, many passes through each trace are needed. In the case of 128Kbyte and 256Kbyte caches, each processor makes up to 20 passes through its trace in a simulation run.

In order to avoid simultaneous cold starts of different processors, which create an unrealistically high bus access activity at the beginning of the bus simulation, each processor is activated at a fixed interval after the previous processor is activated (80K cycles in the simulations). Each trace is then used multiple times by each processor, with the cache being invalidated (not flushed) after a pass through the trace has been completed. This is equivalent to running the same program over again with a new cold start cache. Thus the workload simulates a multiprocessor with a set of steady job streams, each of which has its own characteristics and runs on a different processor.

---

[1]Distinguish between the cold start of a single processor and the ''cold start'' or initial start of the multiprocessor simulation. Simulation must be carried out for sufficient time so that the transients due to the initial start of the multiprocessor have subsided. During a simulation, however, each individual processor may be cold started several times.

Since each processor makes a different number of passes through its trace, depending upon the multiprocessor configuration and the bus activity of a particular trace for the configuration, care must be taken in collecting simulation statistics for each processor. The easiest and most obvious way to collect simulation statistics is to start the collection for each processor at the beginning of the simulation, and stop at the end. However, with this method, the statistics gathered for each processor represent a time period in which a processor may have made a non-integral number of passes through its trace. Because the bus activity can be highly skewed, depending upon the cache organization, it may end up in a situation in which the trace characteristics fed to the model are different from the trace characteristics for the ‘‘equivalent’’ simulation case. This problem can be severe if simulation is not carried out for a long-enough time period so that each processor has made several passes through its trace. To overcome this problem, simulation statistics were gathered for each processor only for a duration that represents an integral number of passes through its trace. In order to maintain a fixed multiprogramming level at all times, every processor is required to run to the end of a simulation, even though some of its activities are not included in its statistics.

For each multiprocessor system, several memory system parameters and instruction execution speeds are considered. Specifically the design space includes: (i) average zero memory-wait-state instruction execution times of 2, 3 or 4 bus cycles[2], (ii) cache sizes of 4K, 8K, 16K, 32K, 64K, 128K and 256K bytes, (iii) cache block sizes of 4, 8, 16, 32, 64, 128, 256 and 512 bytes, (iv) direct mapped and 2-way set associativity in each cache, (v) main memory latencies of 3, 5, 7 and 9 cycles and (vi) circuit switched and STP buses, each with a bus width of 32 bits and multiplexed address and data lines. The cross product of the parameters allows us to evaluate and compare system performance using the trace-driven simulation and the CMVA models with three non-homogeneous and homogeneous workloads, for 5,376 system configurations.

3.4. Comparison of Model and Simulation Results

The primary use of the CMVA models is to predict multiprocessor performance, of which the most important metrics include the total **multiprocessor throughput** and the **total bus utilization**. However, the models first calculate the individual processor throughputs and partial bus utilizations, and then sum them up to get the totals. Therefore I’ll first consider the individual processor results in the following sections, and then comment on the multiprocessor results.

[2]All times in the model are in terms of bus cycles where a bus cycle is the time taken for a single transfer on the bus.

Since three sets of workload are used, and their results are not very different from each other, only the set with non-homogeneous workload of I386 traces will be discussed in detail for all the performance metrics considered. The validation results of the other two sets of workload are then presented for selected metrics.

Before the comparison results from the trace driven simulation and the CMVA models are presented several remarks about what the validation process is expected to accomplish should be made. In particular it should be pointed out out why differences may exist between the results of the modeling and the actual trace driven simulation to justify this validation effort.

Since the analytical models and the simulator have identical assumptions about the multiprocessor hardware, the only reason that the CMVA models may be inaccurate is that the CMVA models do not model the multiprocessor system *workload* as used in simulation accurately in every aspect. In particular, since the service time of the bus with first-come first-served service discipline is deterministic the models have approximated the *mean residual service time* to be half of the service time. The actual mean residual service time, however, can be close to one full service time when a processor has a very high cache miss ratio. When the miss ratio is very high, successive cache misses from the processor have very short inter-arrival times. If the inter-arrival time is shorter than the service time the mean residual service time seen by the cache miss read request should be larger than half of the service time, since the request currently served by the bus can only be started a short while ago (less than the inter-arrival time). The validation process can show if the inaccuracy of this approximation can cause significant difference in the results obtained by the simulation and modeling methods.

Another reason that the CMVA models may yield inaccurate results compared to the trace driven simulation is that the parameters used to characterize the workload for the models may not catch all the significant features of the cache miss traces (or bus access traces) that can affect the performance of a multi. The models are tested with the real traces and a large number of multiprocessor cache configurations, with the hopes of exercising they more extensively.

The validation process is used to find out if the above two factors can have significant effect on the accuracy of the models for the multiprocessor configurations considered. The results of the validation is certainly not sufficient for verifying the models, but should be enough to show how much confidence we can put on the models.

3.4.1. Individual Processor Throughput

Figure 3.3 histograms the percentage difference between the values of individual processor throughput obtained from the trace-driven simulation and the CMVA models. Figure 3.3(a) presents the differences for a circuit switched bus and Figure 3.3(b) histograms the differences for STP buses. Each figure represents 2,688 multiprocessor configurations, half of which use 16 processors and the other half 8 processors. Therefore we have 32,256 cases, each of which represents one processor, for each figure. In the histogram, a negative difference indicates that the value obtained by the CMVA models is less than the value obtained by simulation. Each step in the histogram represents a 2% difference.

From Figure 3.3 we see that in about 75% of the total 64,512 cases the difference in processor throughput obtained from the two techniques is within 1%, and in about 94% of the cases the difference is within 3%. In less than 2% of the cases the difference is larger than 5%, and the largest difference never exceeds 15%. This result shows that the CMVA modeling technique appears to be as accurate as trace-driven simulation in predicting the

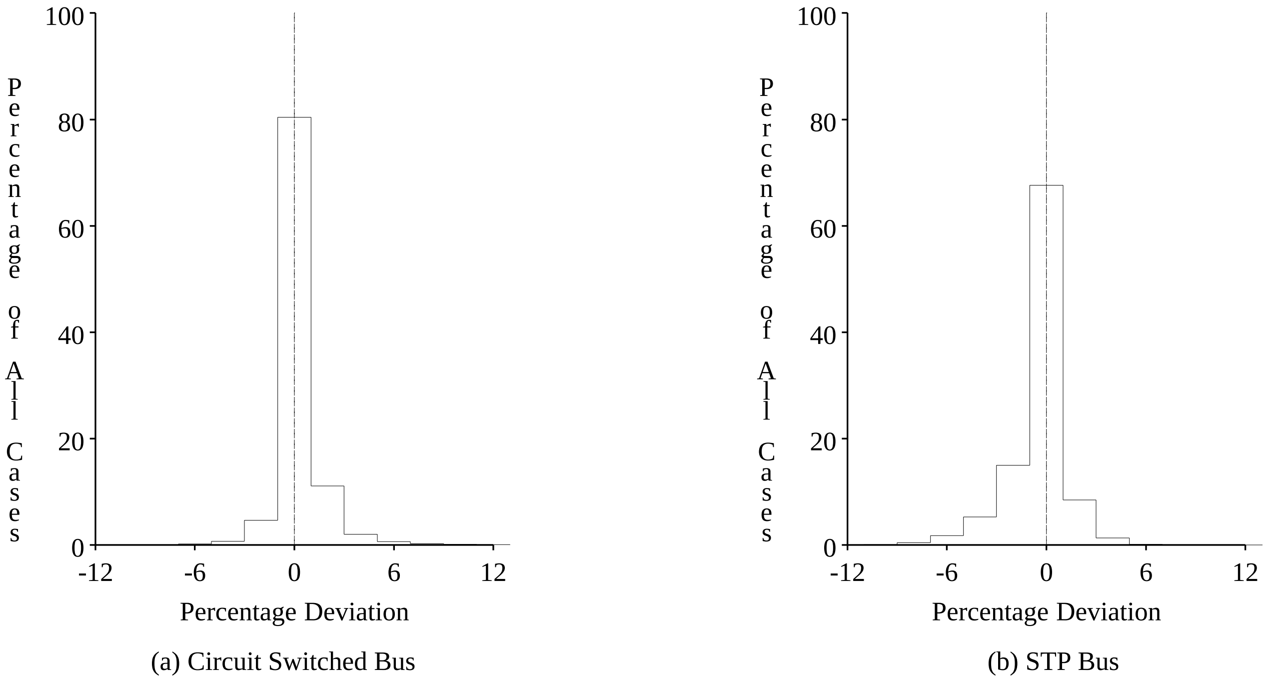


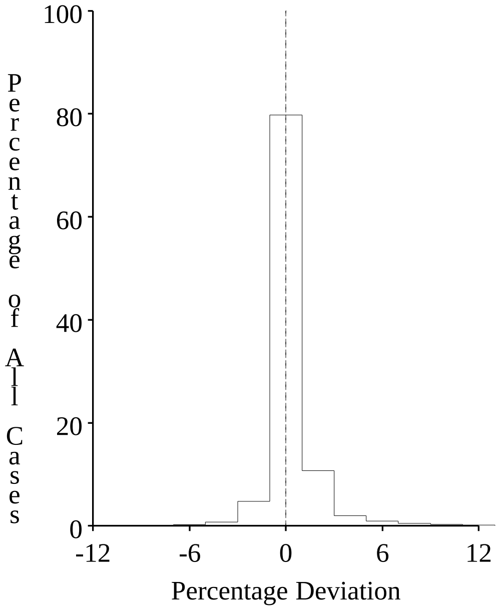
Figure 3.3: Difference in Individual Processor Throughput

individual processor throughputs for a wide range of multiprocessor configurations.

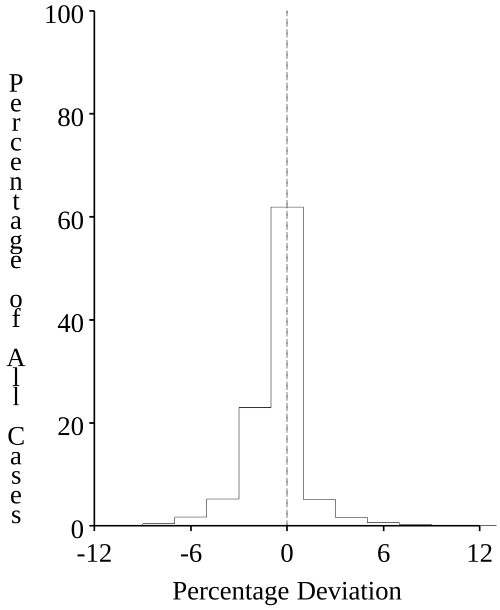
3.4.2. Partial Bus Utilization of Each Processor

We now consider the accuracy of the models in determining the partial bus utilization of each processor. If U_i is the partial bus utilization of a processor and T_i its throughput, then $U_i = D_i \times T_i$, where D_i is the average bus service demand of the processor requests[1]. Theoretically, the percentage difference in the throughput of each individual processor between the model and simulation should be the same as the percentage difference in its partial bus utilization, if the average bus demand D_i is the same for the model and for simulation.

Figure 3.4 histograms the percentage difference in partial bus utilizations of the processors between the values obtained from the models and simulation. From Figure 3.4 we see that in about 70% of the 64,512 cases, the difference is within 1% and in about 92% of the cases the difference in less than 3%. In all cases the differences lie



(a) Circuit Switched Bus



(b) STP Bus

Figure 3.4: Difference in Partial Bus Utilization of Each Processor

---

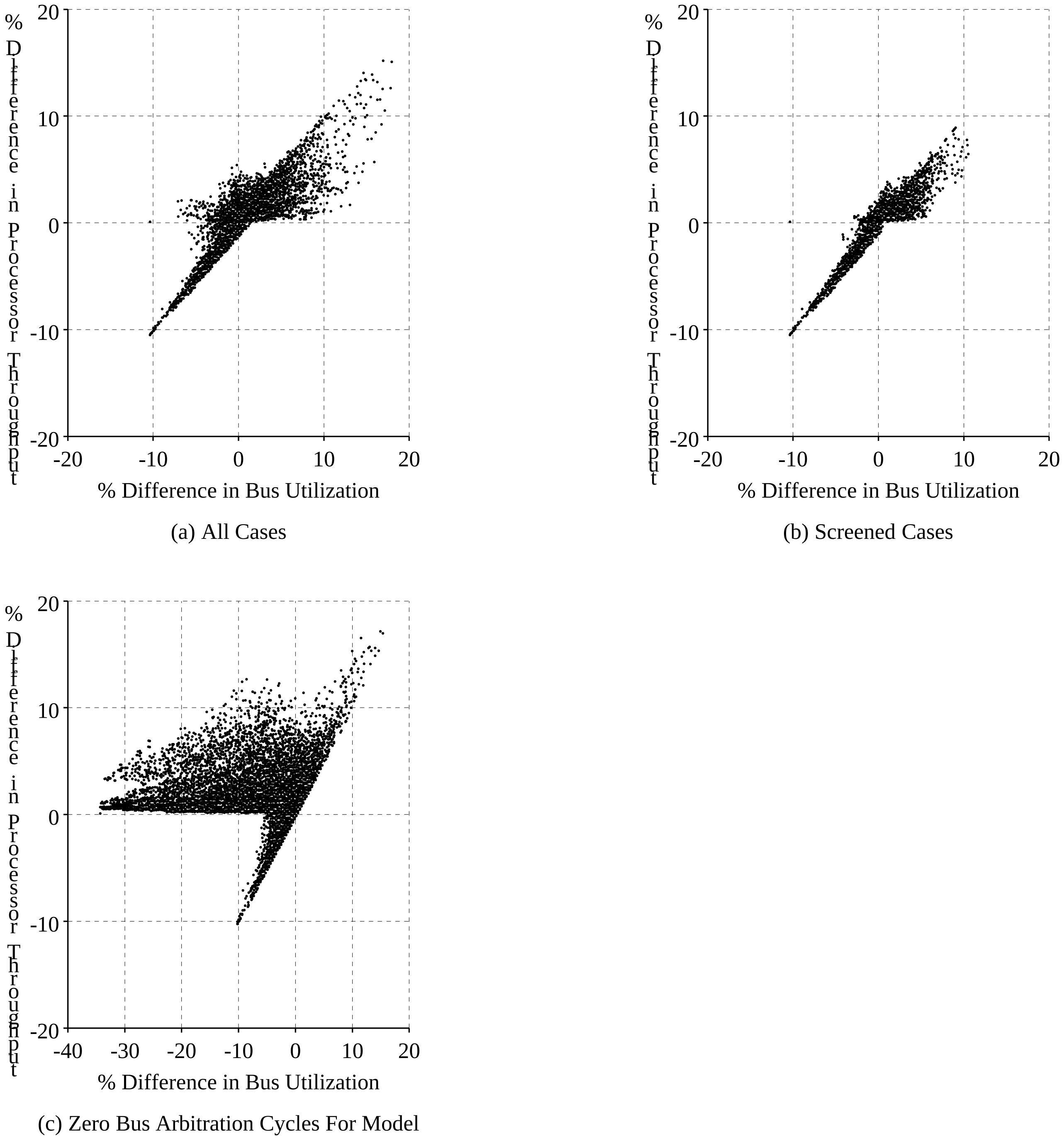
[1] D_i, measured in cycles per instruction, is the product of the misses per instruction and the average number of bus cycles it takes to service a miss.

between -11% and 18%. The result indicates that the CMVA models perform about equally well in predicting individual bus utilization and processor throughput.

If we compare Figures 3.3(a) and 3.4(a), and Figures 3.3(b) and 3.4(b), we see that their shapes are similar, but not exact. That is, the percentage difference in individual processor throughput is not exactly the same as the percentage difference in its partial bus utilization. Figure 3.5(a) emphasizes this fact by correlating the percentage difference between the two metrics. Points that are not on the $X = Y$ line indicate situations where the percentage differences in the metrics differ. The lack of a perfect correlation indicates a possible difference in the average service demand in the model and in simulation. One of the reasons that have been identified which causes the bus service demands to be different in the models and in simulation are the bus arbitration cycles (see the response time equations for the models in an earlier section). Along these lines, a screening of some of the multiprocessor configurations is carried out. Since a bus arbitration cycle is charged at most once to each bus access, accesses that use the bus for a longer period can be expected to be affected less than the cases in which the bus is used for a shorter duration. For example, the bus arbitration cycle, if charged, has a bigger impact on the bus service demand for smaller block sizes than for larger block sizes. By screening out those configurations with cache block sizes of 4 or 8 bytes from Figure 3.5(a), we obtain Figure 3.5(b). In Figure 3.5(b), the percentage differences in throughput and bus utilization are more strongly correlated than in Figure 3.5(a). This suggests a possible difference in the bus demand of a request between the model and simulation caused by the bus arbitration cycle.

If the bus arbitration cycles are a potential source of error in the model, might it be better off to ignore them in the model? To answer this question, results are obtained from the model by ignoring arbitration cycles, and compared with those of simulation, which still charges bus arbitration cycles for bus accesses as before. This comparison is presented in Figure 3.5(c). As is apparent, the modeling results are significantly worse than before, with the model underestimating the bus service demand of a request and consequently underestimating the partial bus utilization and overestimating the processor throughput. Therefore, even though the bus arbitration cycle may be a source of inaccuracy in the CMVA models, taking it into account is still better than ignoring it.

Simulations are also carried out in which bus arbitration cycles were not charged, and results compared to the results of the model excluding bus arbitration. A perfect correlation has been found between the percentage difference in processor throughput and partial bus utilization in the two cases. Because the correlation is simply a straight line, it is not presented in this thesis.



(a) All Cases

(b) Screened Cases

(c) Zero Bus Arbitration Cycles For Model

Figure 3.5: Difference in Throughput vs. Difference in Partial Bus Utilization for Individual Processors

3.4.3. Cache Miss Latency

Figure 3.6 histograms the percentage difference in the average read latency on a cache miss obtained from the CMVA models and the simulation for circuit switched (32,256 cases in Figure 3.6(a)) and STP buses (32,256 cases in Figure 3.6(b)). The results of cache miss latency are less impressive than those of processor throughputs or bus utilizations, but are still quite accurate. In about 90% of all cases the percentage difference is less than 10%, and all differences are within 36%. Comparing Figure 3.6(a) and 3.6(b), it is hard to tell whether the model for circuit switched buses is more accurate than that for STP buses, although the model for a circuit switched bus shows a tendency of underestimating cache miss latency.

3.4.4. Total Multiprocessor Throughput and Bus Utilization

More important than individual processor throughputs and their partial bus utilizations for evaluating the shared bus multiprocessor is the total multiprocessor throughput, and the total bus utilization. In both simulation and modeling, the total multiprocessor throughput, T, is $\sum T_i$, and the total bus utilization, U, is $\sum U_i$. Therefore, it

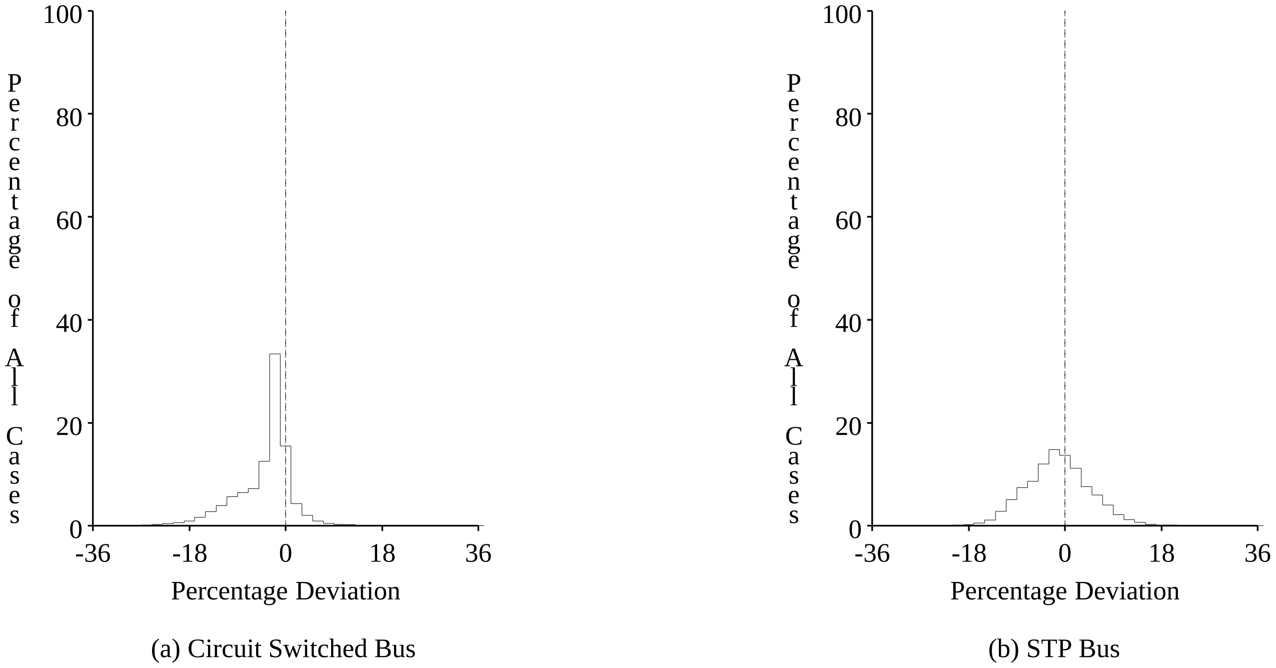


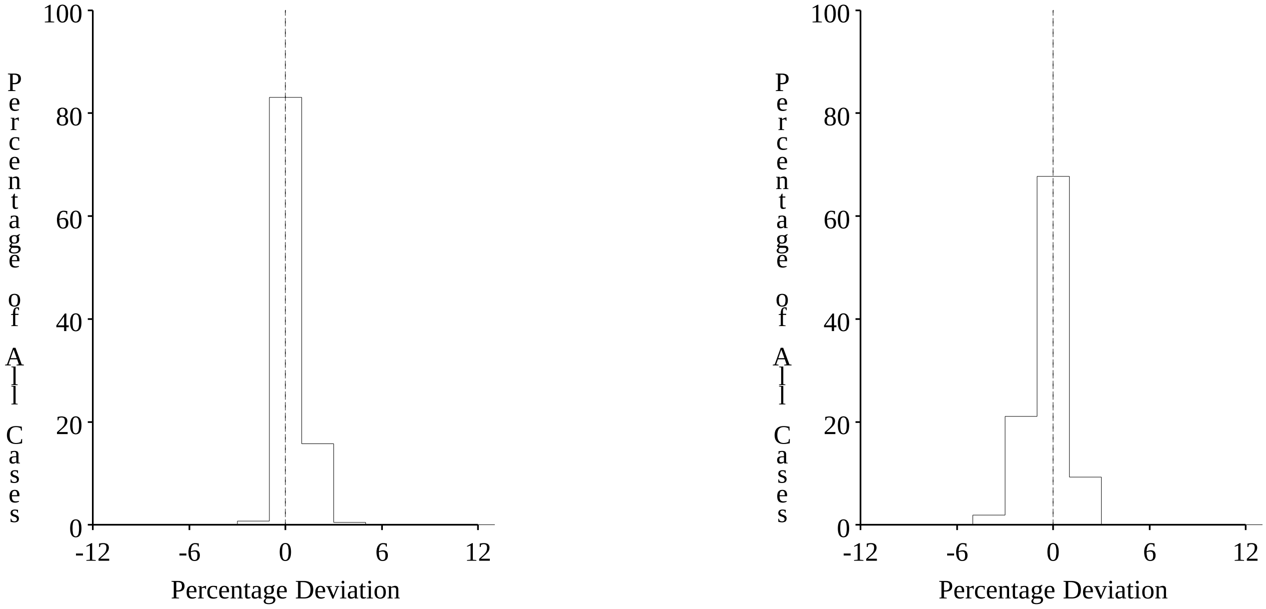
Figure 3.6: Difference in Average Cache Miss Latency of Individual Processors

might be expected that the percentage differences between the results of the models and simulation follow the same trends as in the cases of the individual processors.

In Figures 3.7 and 3.8 the percentage differences between the models and simulation are histogramed for the 2,888 cases with a circuit switched bus, and 2,888 cases with an STP bus, respectively. If Figure 3.3 is compared with 3.7, and Figures 3.4 with 3.8, it can be seen that the models appear to be doing a slightly better job at computing the multiprocessor throughput than at computing the individual processor throughputs, and a slightly worse job at computing the bus utilizations. Moreover, there seems to be a larger discrepancy between the percentage difference in bus utilization and the percentage difference in multiprocessor throughput between the models and simulation than for the individual processor metrics.

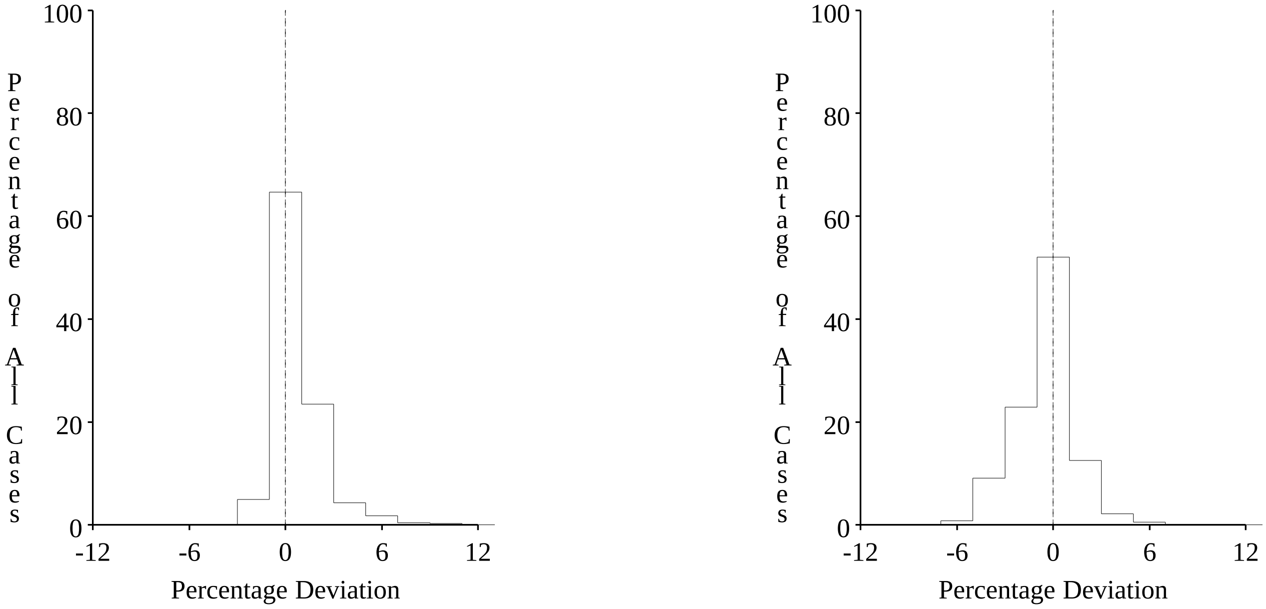
This seemingly incorrect result did initially cause some concern. However, this has a simple explanation, one which is verified with simulation results. Remember that individual processors are executing tasks with different trace characteristics (different misses per instruction and different bus service demands), and do indeed have different throughput and partial bus utilization values in a particular multiprocessor configuration. The processors that have a higher individual throughput will contribute more heavily to the multiprocessor throughput. Likewise, the processors with a higher partial bus utilization will contribute more heavily to the total bus utilization. In the experiments, for the same multiprocessor configuration, the processors with a higher throughput achieved it with more in-cache computation and had a lower partial bus utilization (because they were running a trace with a lower bus demand) than processors with a lower throughput (which were running traces with a higher bus demand). Moreover, the processor(s) with the higher bus demand had a bigger percentage error than the processors with a lower bus demand.

When the individual processor throughputs and partial bus utilizations are summed up to obtain the multiprocessor throughput and bus utilization, the percentage difference between the multiprocessor throughput and bus utilization can be different from the percentage difference between the individual processor throughputs and their partial bus utilizations. This point is best illustrated by means of an example. Consider a multiprocessor configuration in which one processor (processor A) is running a job that misses frequently in the cache, and therefore has a high bus service demand, and a low throughput. The other processors are running jobs that mostly hit in the cache and consequently have a low bus service demand and a high throughput. If there is a large percentage error in computing the throughput and partial bus utilization of processor A, and a small percentage error in computing these values



(a) Circuit Switched Bus (b) STP Bus

Figure 3.7: Difference in Total Multiprocessor Throughput



(a) Circuit Switched Bus (b) STP Bus

Figure 3.8: Difference in Total Bus Utilization

for the other processors, summing the individual throughput and utilization values will result in a smaller percentage difference in multiprocessor throughput and a larger percentage difference in total bus utilization. Because of this, multiprocessor throughput and bus utilization results should be used with caution if the individual processor bus service demands, and the throughputs and partial bus utilizations differ greatly.

3.4.5. Comparison Results from TITAN Traces

The results of using TITAN traces are shown in Figure 3.9, for the percentage difference in individual processor throughput obtained from the CMVA models and simulations. Figure 3.9 histograms the comparison results for the same 5,376 multiprocessor configurations as before, and therefore a total of 64,512 cases for the two bus switching methods.

The accuracy of the models in predicting individual processor throughput as depicted in Figure 3.9 is comparable to that in Figure 3.3, which is calculated from the I386 traces. Although the number of cases where the percentage difference is within 1% is lower (about 60% of the cases, compared to 70% in Figure 3.3), in about the same number of cases (92% of the cases v.s. 94% in Figure 3.3) the percentage difference is within 3%. The comparison results from the TITAN traces are more accurate in the maximum percentage difference. In Figure 3.9 the maximum difference is less than 10%, as opposed to 15% in Figure 3.3. So the CMVA models developed here can predict the individual processor throughput for the RISC (Titan) as well as for the CISC (Intel 386) processors.

3.4.6. Results from Homogeneous Workload

The comparison results of individual processor throughput for the homogeneous workload, is shown in Figure 3.10. As in Figure 3.3 and Figure 3.9, a total of 64,512 cases of comparisons are represented in Figure 3.10. Compared to Figure 3.3, the number of cases where the percentage difference in individual processor throughput are within the 1% and 3% ranges are similar (67% and 91% respectively in Figure 3.10). The maximum difference, however, is slightly better (less than 11% in Figure 3.10). So the CMVA models are about the same accuracy, whether homogeneous or non-homogeneous workload is used.

It should be noted that differences in the values obtained from the CMVA and trace-driven simulation methods are not entirely caused by the inaccuracy of the CMVA models. The differences are partly due to *imperfect simulation process*. This is true no matter homogeneous or non-homogeneous workload is used. But the factor of imperfect simulation process becomes more apparent when homogeneous workload is used. With homogeneous

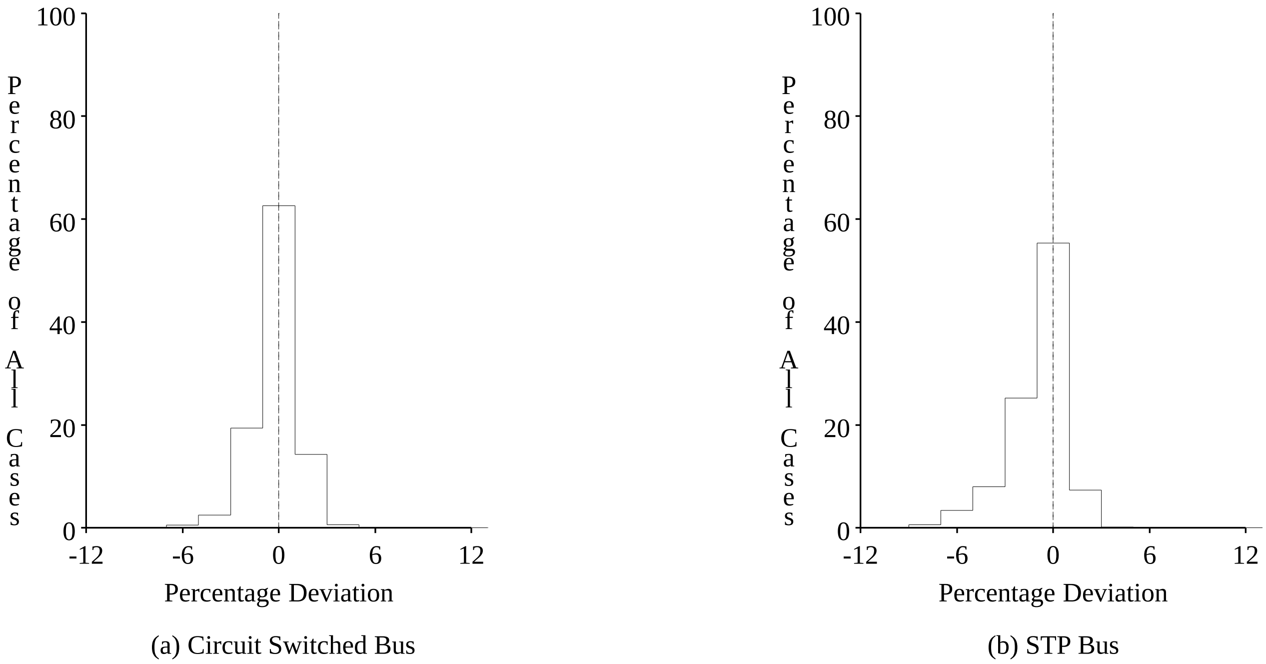


Figure 3.9: Difference in Individual Processor Throughput; Using TITAN Trace

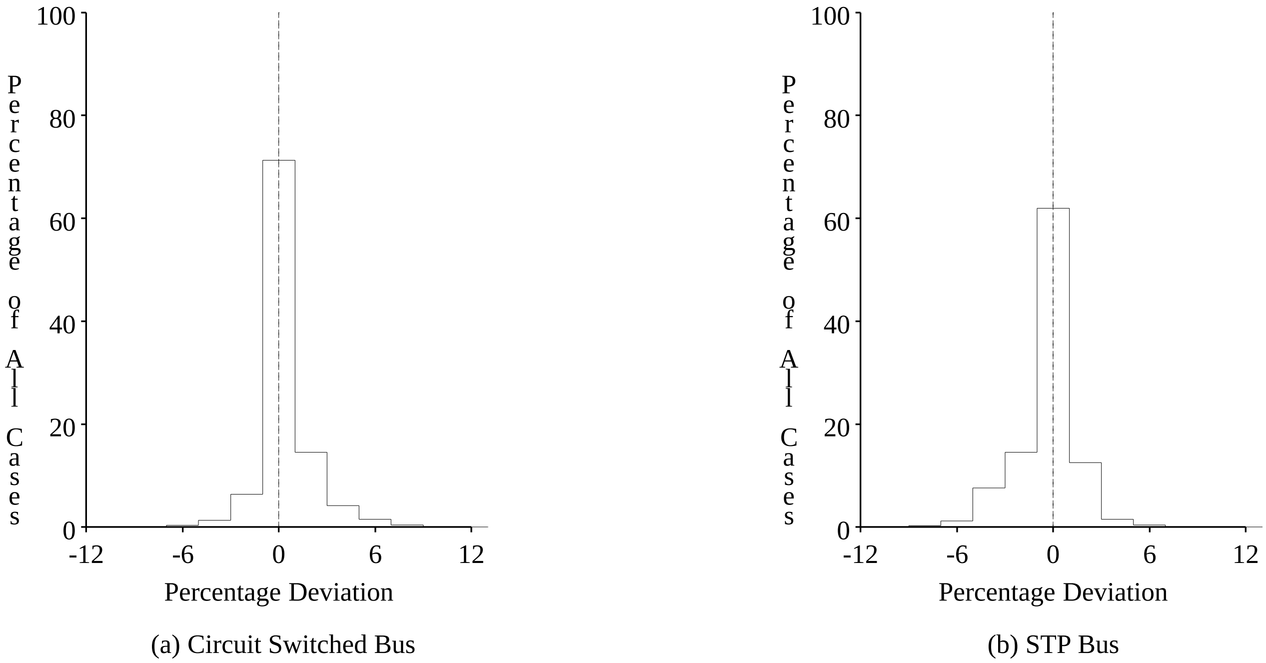


Figure 3.10: Difference in Individual Processor Throughput; Homogeneous Workload

workload the analytical models duly generate the same performance statistics for every processor in the same configuration. In simulation, on the other hand, processors proceed in a different pace from each other at different times, even though the same set of benchmark programs (though run in different order) is used. If simulation can run for a very long time the performance results from all processors should converge to the same values. But with limited simulation time, the values derived are slightly different for different processors. Therefore the accuracy of the CMVA models should be better than reflected from the comparison results in Figure 3.3, Figure 3.9 or Figure 3.10.

The comparison results of the other performance metrics for the TITAN traces and the homogeneous workload have also been obtained. They are not shown here because they do not suggest any significantly different conclusion about the accuracy of the models than what has been established in the above discussion.

3.5. Summary and Conclusions

In this chapter the experience with using mean value analysis models for analyzing shared bus, throughput-oriented multiprocessors is reported. The values of three performance metrics, namely the processor throughput, the bus utilization and the cache miss latencies obtained from the CMVA models are compared with the corresponding values obtained from an actual trace-driven simulation for 5,376 system configurations.

The results are quite encouraging. With a non-homogeneous workload of Intel 386 traces the comparison results indicate that the CMVA models are able to estimate individual processor throughput to within 3% for about 94%, and individual processor bus utilization for about 92%, of all cases. Estimation of the cache miss latencies is less accurate than the estimation of processor throughput or bus utilization, but is still quite good. A potential source of the inaccuracy has been identified. It is related to bus arbitration cycles, and can be a significant portion of bus utilization when the cache block size is small.

The comparison results derived from another set of non-homogeneous workload of TITAN traces and a homogeneous workload of Intel 386 traces also demonstrate the same accuracy of the CMVA models. In particular, the estimate of individual processor throughput by the models is within 3% in about 92% of all cases for the TITAN traces, and 91% for the homogeneous workload. The comparison results from homogeneous workload also makes apparent that, the difference in the values derived from the models and the trace-driven simulation is at least partially due to imperfect simulation process. Thus the accuracy of the models may still be better than what has been suggested by the results presented here.

Because the execution time of the CMVA model is about 4 to 5 orders of magnitude less than trace-driven simulation, and because the results of the two approaches are in close proximity, the CMVA models are an excellent choice for exploring the design space for shared bus, throughput-oriented multiprocessors.

Chapter 4
Evaluation of Design Choices for Memory System in a Throughput-Oriented Environment

4.1. Overview

The CMVA models developed in Chapter 3 are quite accurate over a wide range of multiprocessor configurations. Since their solution is 4 to 5 orders of magnitude faster than trace-driven simulation, the models are used for the evaluation of design choices. The models require inputs of M, M_{ref}, P_r, P_w and P_v (all terms are defined in Section 3.2.2). The values of these inputs should be obtained from traces that are representative of the workload for which design choices are evaluated. While the miss ratio characteristics, i.e., M, of various cache organizations are easily available from the literature for a wide variety of workloads, the values of M_{ref}, P_r, P_w and P_v are typically not available.

Since the results of evaluating design choices may depend very much on the workload, it is important to choose workloads that are both realistic and representative. The traces used in Chapter 3 to validate the models could be re-used. The two kinds of traces, in fact, represent the two broad categories of processor architectures. The Intel 386 traces are derived from CISC 80386 processors, while the TITAN traces from RISC Titan processors. Using workloads from these two distinct architectures would make the evaluation of design choices a fairly complete study.

However, for CISC architectures better traces are publicly available. Traces generated using the Address Tracing Using Microcode (ATUM) technique are more representative because they contain operating system activity [2]. Moreover, to put the evaluation results in perspective, the workloads should have been used previously for uniprocessor cache studies. The ATUM traces, as well as the TITAN traces, fulfill this requirement.

In the ATUM technique, patches are made to the microcode of the machine to generate addresses for all the memory references made by the processor. These references include references made by user programs as well as references made by the operating system. The ATUM traces that we use are gathered via microcode patches on a VAX 8200 by Agarwal and Sites. These traces are distributed by DEC, are considered to be the best public-domain traces for a multiprogrammed, multi-user environment, and they have been widely used in recent cache studies [3, 5, 34, 35, 56]. By passing the ATUM traces through a uniprocessor cache simulator we obtain the values of M,

M_{ref}, P_r, P_w and P_v.

The microcode based ATUM technique obviously can not be applied to RISC processors. Although the software method used in collecting the TITAN traces is capable of extracting memory trace from operating system functions, the traces currently available do not include any operating system activity [15]. Nevertheless the TITAN traces still represent the memory access behavior from a state-of-the-art, new generation RISC architecture.

Keep in mind that the goal is to evaluate the impact of a particular design choice in the memory system on the peak multiprocessor throughput that can be supported by the memory system, or the *maximum multi throughput* for the memory system configuration (cache, shared bus and main memory). This is done using the following procedure. For each memory system configuration, the total multi throughput (which is the sum of the throughputs of each processor in the multi) is computed for an increasing number of processors. The maximum multi throughput is the throughput at the point beyond which the addition of more processors contributes less than 1% to the total throughput of the multi, i.e., the throughput when the bus is saturated. The exact number of processors in the multi at the point at which the maximum multi throughput is achieved varies with the parameters of the memory system.

Unless mentioned otherwise, for all the system configurations that are evaluated in the coming sections, the bus is assumed to be 32 bits wide with multiplexed address/data lines and has a cycle time of 50ns (or 20 MHz), the processor CPUs have a peak performance of 5 VAX MIPS for the ATUM traces and 20 Titan MIPS for the TITAN traces, and all caches are assumed to be write back. In all experiments, throughput is measured in VAX MIPS when the traces are relevant to VAXen, and Titan MIPS when the traces are from Titan processors.

It should be noted that the accuracy of the CMVA modeling results depends much on the accuracy of the input workload parameters. The values of input workload parameters are calculated from cache simulation. To collect accurate statistics of a cache simulation both the length and the ‘‘breadth’’, or the distinct memory locations that are actually accessed, of the trace in general should be much larger than the cache size. The VAX traces contain totally about 2.6M memory accesses. The TITAN traces used are the same as used in [15], and contain a total of more than 500M memory accesses. The biggest caches of which the performance is evaluated by the VAX traces are limited to 256K bytes. The TITAN traces, on the other hand, are used to evaluate the caches as large as 1M bytes.

In the following sections the evaluation results of using the VAX traces will be presented first and explained in detail. The results of using TITAN traces will then be given, with more emphasis on those results that are different, or not available from the results of the VAX traces.

4.2. Results From ATUM Traces

4.2.1. Cache Performance Metrics and Uniprocessor Performance

Before the design tradeoffs are evaluated, the performance of several uniprocessor cache organizations using the ATUM traces and traditional uniprocessor cache performance metrics are considered. This allows the design choices for the multiprocessor memory system to be compared with equivalent choices for a uniprocessor memory system.

4.2.1.1. Cache Performance Metrics

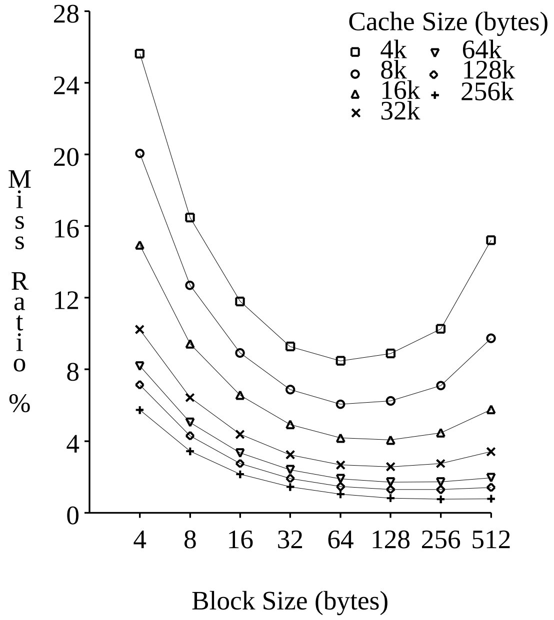
The miss ratio (in percentage) is presented in Figure 4.1(a) for various cache sizes and block sizes (all caches are direct mapped and write back). For bus traffic[1], *data only* traffic (Figure 4.1(b)) is distinguished from *data and address* traffic (Figure 4.1(c)). The data traffic includes only the actual data transfer cycles whereas the address and data traffic also includes the addressing overhead (the bus is 32 bits with multiplexed address and data lines). The data traffic ratio (in percentage) is the ratio of the traffic that appears on the bus in the presence of a cache to the traffic that appears without the cache. Thus, if the data traffic ratio is 400%, it means that the traffic on the bus with the cache is 4 times as much as the traffic without the cache.

As mentioned earlier, the impact of the memory system on processor performance is directly governed by equation (2.2). In a uniprocessor, if T_C^P and T_m^C are independent of the cache organization, the best cache organization is one that minimizes the overall miss ratio. However, as mentioned earlier, T_C^P and T_m^C are not independent of cache organization, and to evaluate the impact of the entire memory system on processor throughput, the impact of T_C^P and T_m^C must be considered. This is illustrated in Figures 4.2 and 4.3.

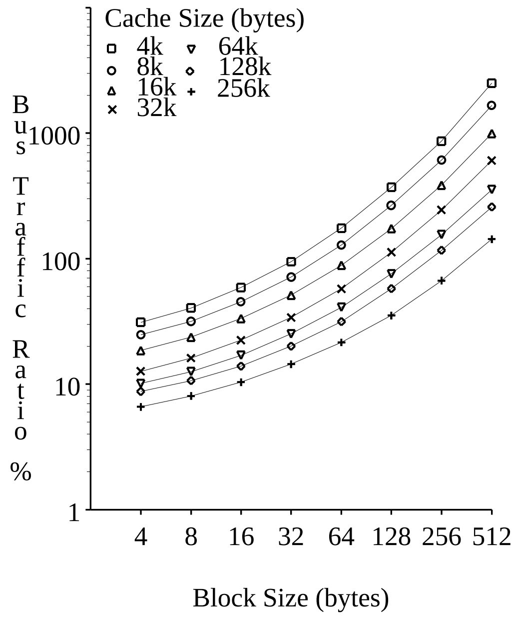
4.2.1.2. Uniprocessor Performance

In Figure 4.2, the throughput of a uniprocessor (in VAX MIPS) is plotted as a function of the main memory latency for several cache sizes and main memory latencies. For all cases, the cache is direct mapped. The trends to be observed from Figure 4.2 are somewhat obvious: i) as the main memory latency increases, T_m^C increases and

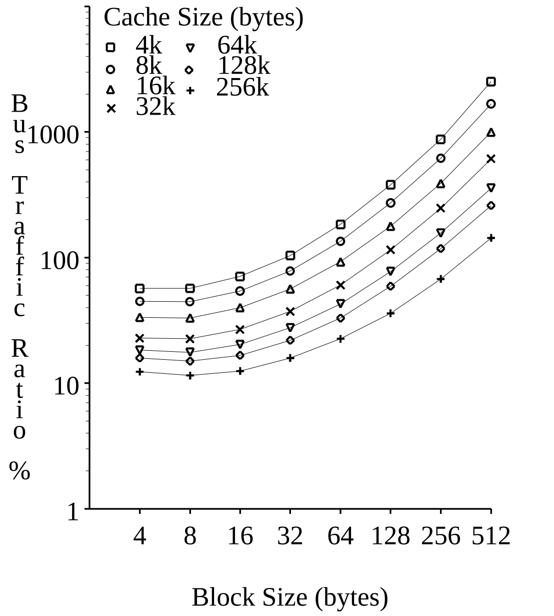
[1]Bus traffic includes traffic generated to service miss requests, as well as write back and invalidation requests.



(a) Miss Ratio

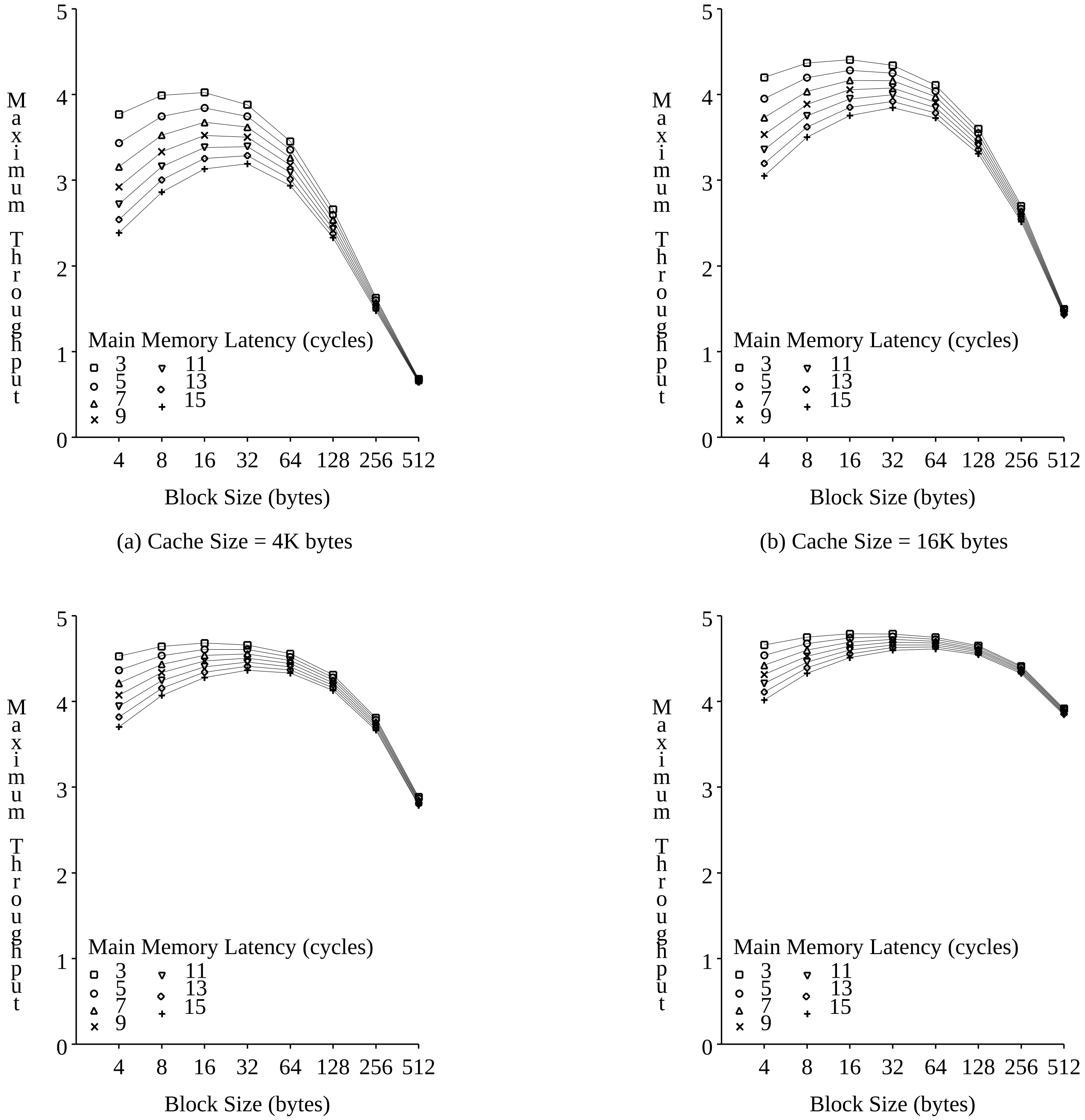


(b) Bus Traffic Ratio (Data Only)



(c) Bus Traffic Ratio (Data and Address)

Figure 4.1: Cache Performance Metrics for the ATUM Traces



(a) Cache Size = 4K bytes

(b) Cache Size = 16K bytes

(c) Cache Size = 64K bytes

(d) Cache Size = 256K bytes

Figure 4.2: Uniprocessor Throughput (in VAX MIPS) with Varying Main Memory Latency

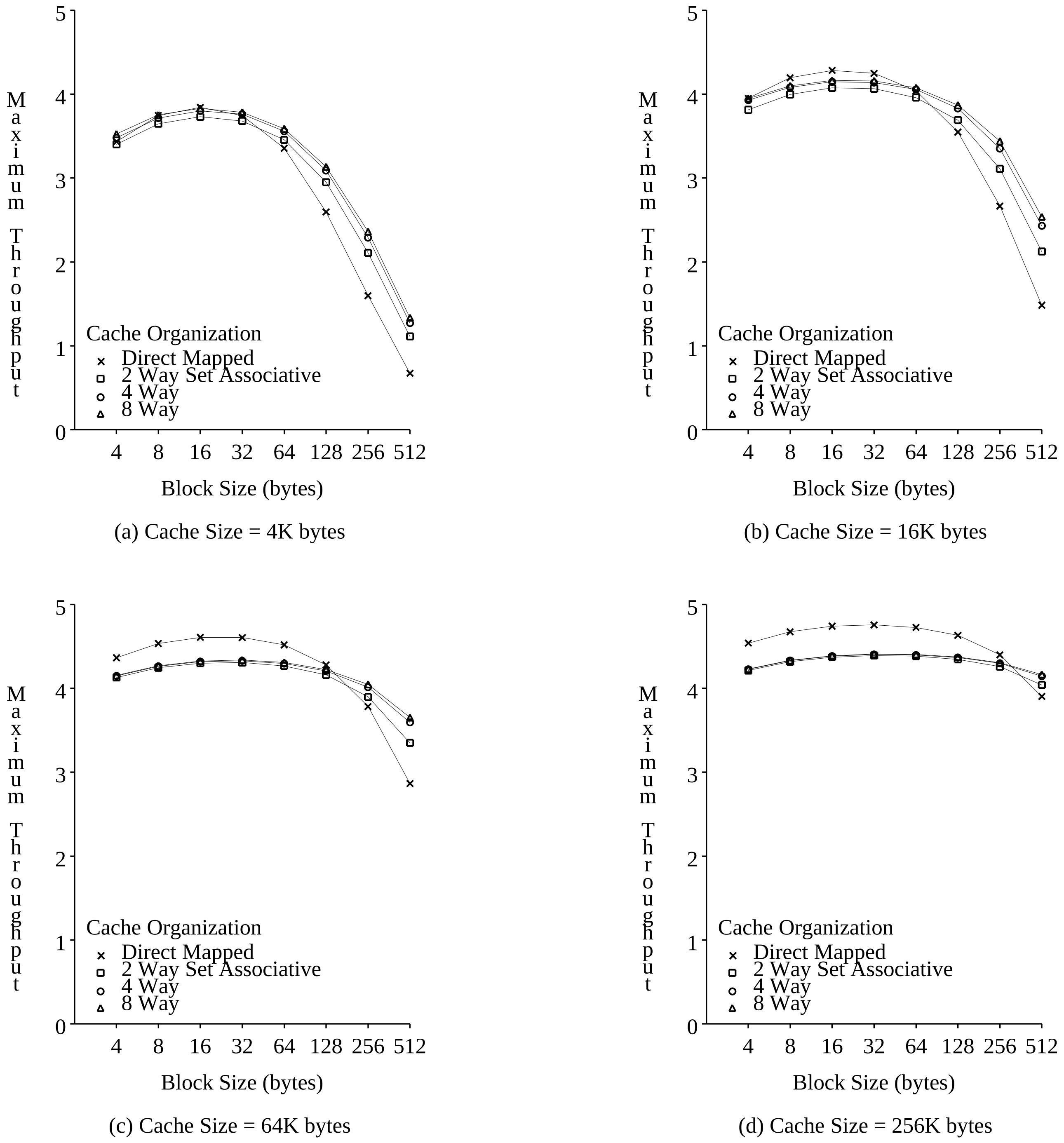


Figure 4.3: Uniprocessor Throughput (in VAX MIPS) with Varying Cache Set Associativity

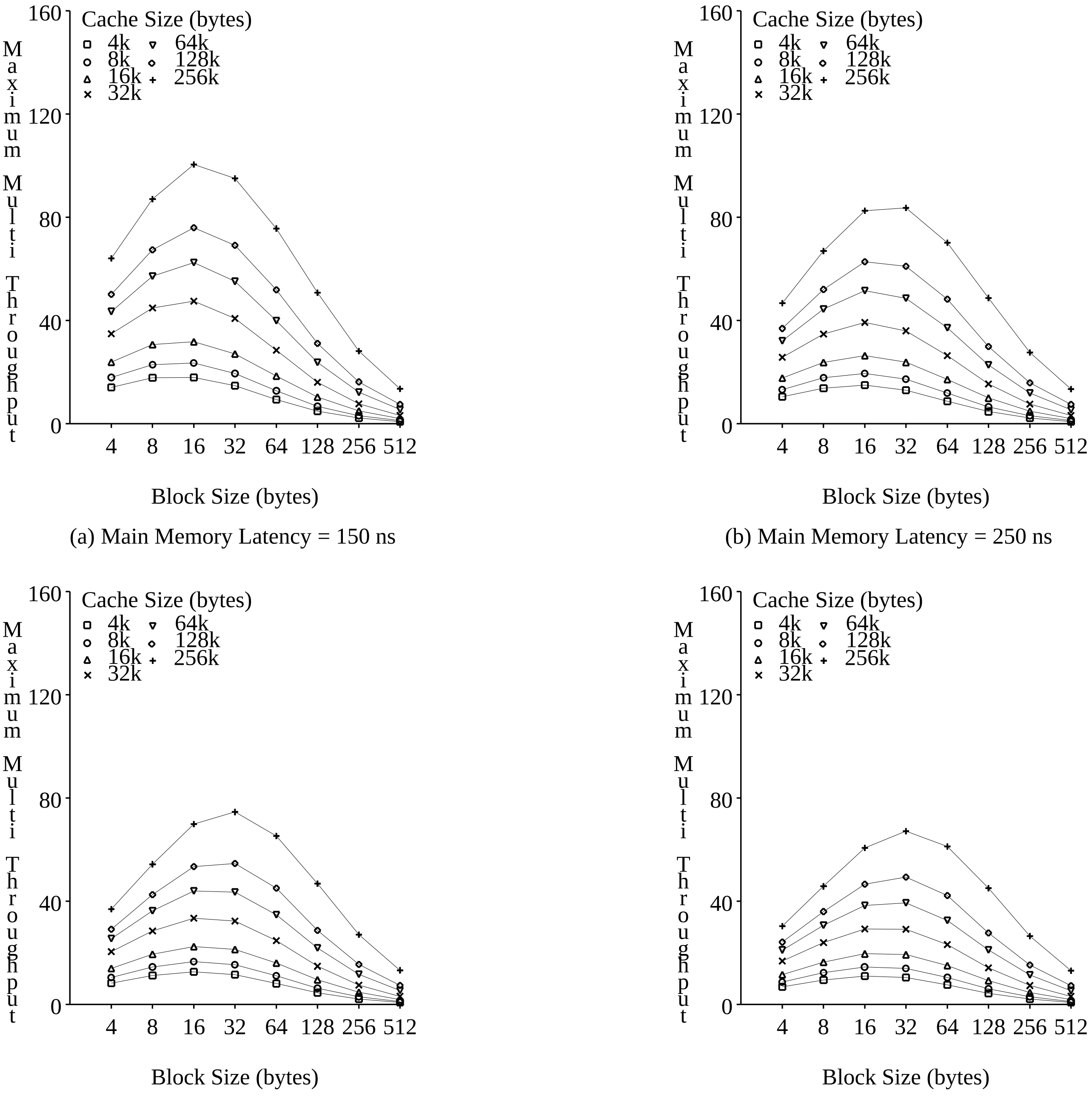
consequently the throughput of the uniprocessor decreases, and ii) the impact of the main memory latency on processor throughput is sensitive to the cache block size. As will be seen, in multiprocessors neither trend needs to be as pronounced as in the case of uniprocessors, especially with an STP bus. More on this in Section 4.2.2.

Figures 4.3(a)-(d) plot the processor throughput for cache sizes of 4K, 16K, 64K and 256K bytes, respectively, each with varying set associativity and block size. The main memory latency is kept fixed at 250ns (5 cycles) in all cases. To account for the impact of set associativity on processor throughput, T_C^P of caches with set associativities of 2, 4 and 8 is 10% greater than T_C^P for a direct mapped cache [34]. Two trends are obvious from Figure 4.3. The first trend is that as cache size increases, the block size that results in the best uniprocessor throughput increases. Furthermore, the throughput tends to ‘‘flatten’’ out, indicating that several block sizes may give roughly the same performance. The second trend to note is that as cache size increases, the need for set associativity decreases. For larger caches, when the cycle time advantages of direct mapped caches are taken into account, direct mapped caches can actually provide better throughput than set associative caches, even though the set associative caches may have better miss ratios. Both trends apparent in Figure 4.3 are well known and have been described in detail in the literature on uniprocessor caches [35, 62]. The purpose of presenting them here is again to show that neither trend may occur for multiprocessor caches, to be discussed in the upcoming sections.

4.2.2. Cache Block Size Choice

To evaluate the choice of a block size, only direct mapped caches are considered (other set associativities are considered in Section 4.2.3). Using the CMVA models, the maximum multi throughput is calculated as the block size is varied for different cache sizes and main memory latencies. Figures 4.4(a)-(d) present the maximum multi throughput (in VAX MIPS) with various cache sizes and main memory latencies for a circuit switched bus and Figures 4.5(a)-(d) present the same for an STP bus.

From Figure 4.4, several observations are made about memory system design choices with a circuit switched bus. First, larger block sizes tend to be favored as the cache size is increased. However, the trend towards larger block sizes is not as strong as in the case of a uniprocessor (compare Figure 4.4 with Figure 4.2). While the trend towards larger block sizes may seem obvious, it should be pointed out that this conclusion can not be derived from a simple consideration of the bus traffic and/or the miss ratio cache metrics. Figure 4.1 shows that the miss ratio metric favors larger block sizes as cache size is increased, but the bus traffic metric still favor smaller block sizes. In a circuit switched bus, consideration of the bus traffic alone is clearly not sufficient since the bus is held by the



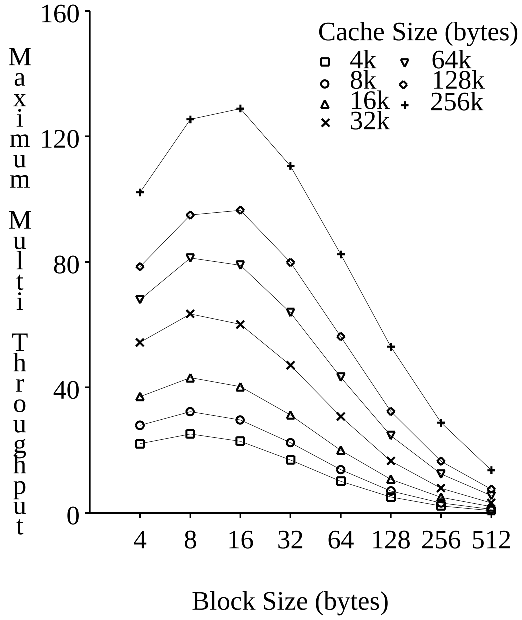
(a) Main Memory Latency = 150 ns

(b) Main Memory Latency = 250 ns

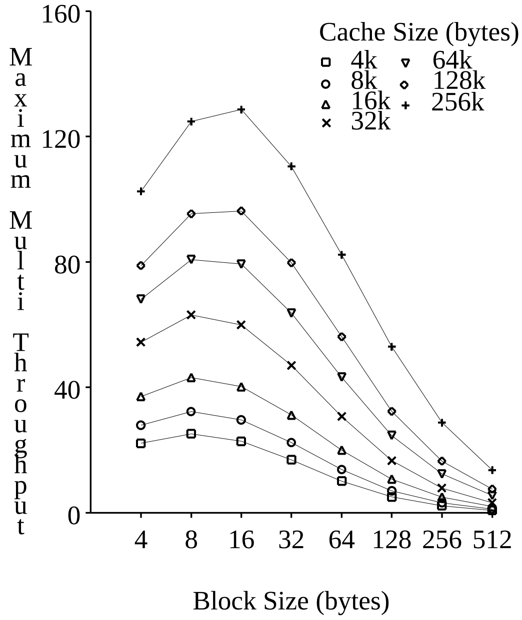
(c) Main Memory Latency = 350 ns

(d) Main Memory Latency = 450 ns

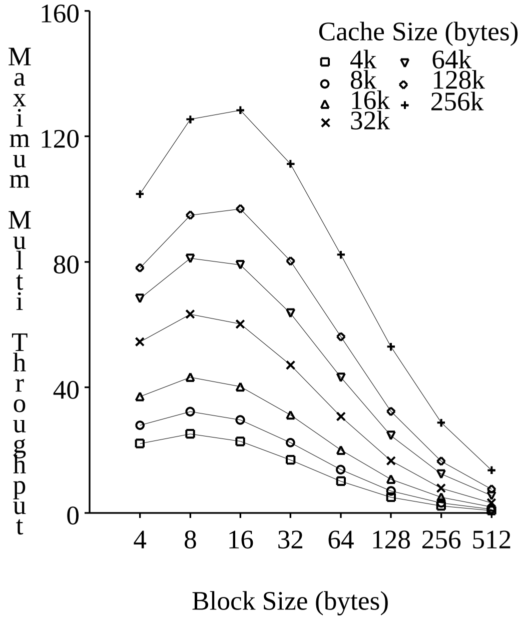
Figure 4.4: Maximum Multi Throughput (in VAX MIPS) with a Circuit Switched Bus



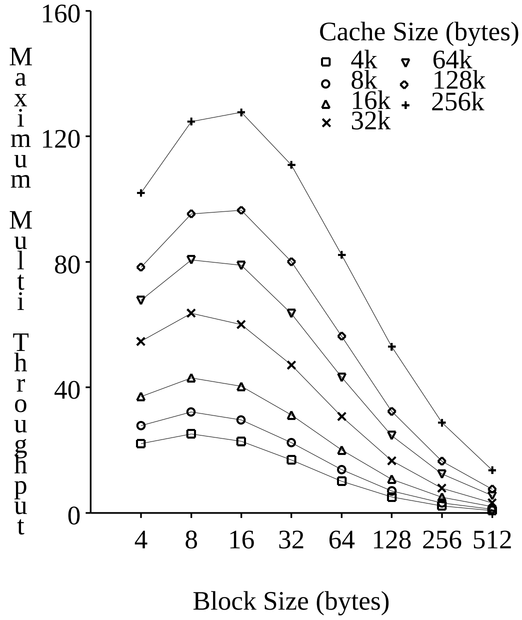
(a) Main Memory Latency = 150 ns



(b) Main Memory Latency = 250 ns



(c) Main Memory Latency = 350 ns



(d) Main Memory Latency = 450 ns

Figure 4.5: Maximum Multi Throughput (in VAX MIPS) with an STP Bus

master until the entire transaction is complete. A read transaction includes the main memory latency and therefore, the data traffic performance metric (which is influenced only by the cache organization and not by other parameters of the memory system) is not a good indicator of the bus utilization. To accurately evaluate design choices, all factors that can influence performance must be taken into account.

Second, the choice of the block size is also sensitive to the main memory latency in a circuit switched bus. When main memory latency is high, larger block sizes tend to be favored (Figure 4.4), just as in the case of a uniprocessor (Figure 4.2).

Third, the main memory latency has a significant impact on the maximum performance that can be achieved. For example, going from a main memory latency of 150ns to 450ns with a 256K byte cache, the maximum multi throughput, with the best block size, decreases from about 100 MIPS to about 67 MIPS. This is because the communication protocol of a circuit switched bus is such that the bus is not available for use until the entire transaction is complete, and a large main memory latency contributes significantly to the bus utilization.

For an STP bus (Figure 4.5), the results are somewhat different. First, larger block sizes seem to be favored as cache size increases (up to a point), just as in the case of a circuit switched bus. In an STP bus, the bus traffic (address plus data) is an accurate indicator of the utilization of the bus. Therefore, why might larger block sizes be favored with STP buses even though smaller block sizes result in a lower bus utilization? To understand this, we need to look at equation (2.2) as well as the shapes of the miss ratio and the traffic ratio in Figure 4.1.

The memory latency in equation (2.2), T_m^P, includes both the probability of making a bus request (the miss ratio M) as well as the queuing delay that the request experiences (a part of T_m^C). While the queuing delay increases as the utilization of the bus increases with a larger block size, M may decrease sufficiently with the larger block size to offset the additional queuing delay. That is, with a larger block size, the processor may be able to achieve a higher throughput by carrying out local (in cache) computation more often than with a smaller block size, even though it experiences a bigger penalty for non-local access. The opposing trends in miss ratio and bus traffic (or bus utilization in case of an STP bus) in Figure 4.1 lead to a best block size that may not result in either the best miss ratio or the best bus traffic.

Second, the choice of the block size that allows the best maximum multi throughput seems to be insensitive to the main memory latency. In fact, this is just one facet of a more interesting phenomenon that the maximum multi throughput appears quite insensitive to the main memory latency.

These seemingly counter-intuitive observations can be explained as follow. If we view a main memory reply in response to an earlier memory read from a processor as part of the bus access activity of the processor, increasing the main memory latency has the same effect as increasing the idling time between the two accesses (the bus read and the subsequent main memory reply). The resulting smaller bus access rate of each processor (due to the increased idling time) reduces the bus utilization and hence the bus queuing delay. Therefore, the cache miss latency, T_m^C, which includes the main memory latency as well as the queuing delay, does not increase to the same extent as the increase in the main memory latency. Figure 4.6 shows this effect. In the initial configuration the main memory latency is 3 cycles (or 150ns) and each processor has a 64K byte cache. By connecting a sufficient number of processors to the bus, the system saturates and delivers its maximum throughput. Keeping the same number of processors that saturate the bus with a main memory latency of 3 cycles, the increase in cache miss latency is plotted against larger values of main memory latency. From Figure 4.6 we can see that, for example, when the block size is 64 bytes, changing the main memory latency from 3 cycles to 15 cycles (750ns) increases the cache miss latency by only 2 cycles (100ns); the difference represents a decrease in the queuing delay because of the slightly slower bus access rate of each processor, and the consequent lower bus utilization.

The increase in miss latency, however reduced, still decreases the throughput of an individual processor. However, since the bus utilization is also reduced, more processors can be added to compensate for the loss of performance of the individual processors. This is illustrated in Figure 4.7 which shows the number of processors used to deliver the maximum multi throughput for different main memory latencies. As we can see, the number of processors that can be connected together to achieve the maximum multi throughput increases with the decrease in throughput of each processor due to the increase in main memory latency. Putting it together, the maximum throughput of a multi with an STP bus seems to be quite insensitive to the main memory latency, as evidenced by the nearly identical graphs for varying memory latencies in Figures 4.5(a)-10(d). Of course, if the number of processors in the multi were fixed, the throughput of the multi would decrease as the memory latency is increased.

4.2.3. Cache Set Associativity Choice

Now the choice of the set associativity for the cache in a multi is considered. In Figures 4.8(a)-(d), the maximum multi throughput is presented that can be supported by a memory system using cache sizes of 4K, 16K, 64K and 256K bytes, respectively, with varying set associativities. For each cache size, direct mapped, 2-way, 4-way and 8-way set associative organizations are considered. Again the cycle time of a cache with set associativity of 2,

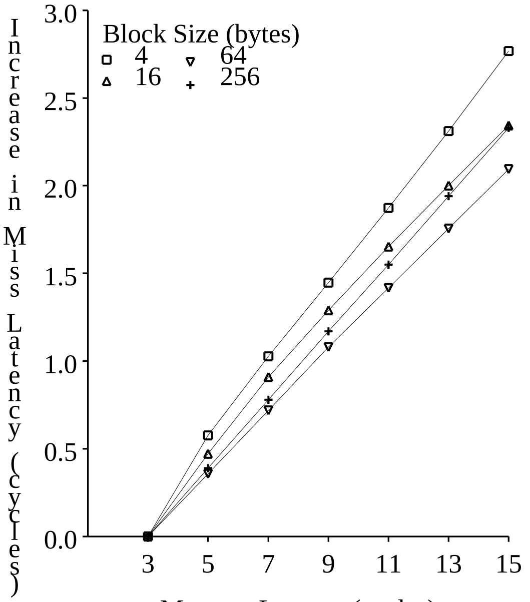


Figure 4.6: Increase in Cache Miss Latency

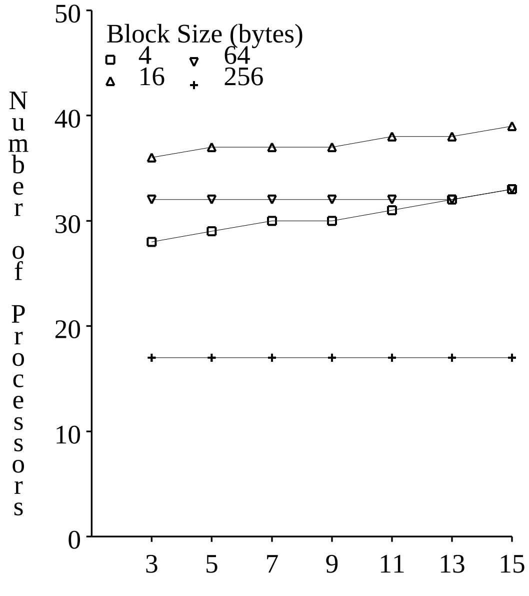


Figure 4.7: No. of Processors that Give the Maximum Multi Throughput

4 or 8 is assumed to be 10% longer than the cycle time of a direct mapped cache. The bus is an STP bus and the main memory latency is 250 ns (5 cycles).

In Figure 4.8 we can see that for all four cache sizes, the maximum multi throughput increases at least 20% when 2-way set associative instead of direct mapped caches are used, if these caches always choose the block sizes that give the best performance. For example, when cache sizes are 256K bytes and block sizes are 16 bytes, the maximum multi throughput increases from 128 MIPS with direct mapped caches to 156 MIPS with 2-way set associative caches, an improvement of 22%. These data suggest that 2-way or 4-way set associativity may be warranted in a multi even when the cache size is fairly large (256K bytes). This is unlike uniprocessor caches where the need for set associativity diminishes significantly as the cache size increases [34, 35, 56].

The reason why a larger associativity is favored for the multiprocessor caches is due to the fact that caches with a larger associativity lower the miss ratio as well as the per-processor utilization of the shared bus. The lower bus utilization results in a lower queuing delay and consequently a lower overall T_m^C. Therefore, the product $M \times T_m^C$ might decrease sufficiently to offset the increase in T_C^P, resulting in a lower T_m^P. This is in contrast to a uniprocessor

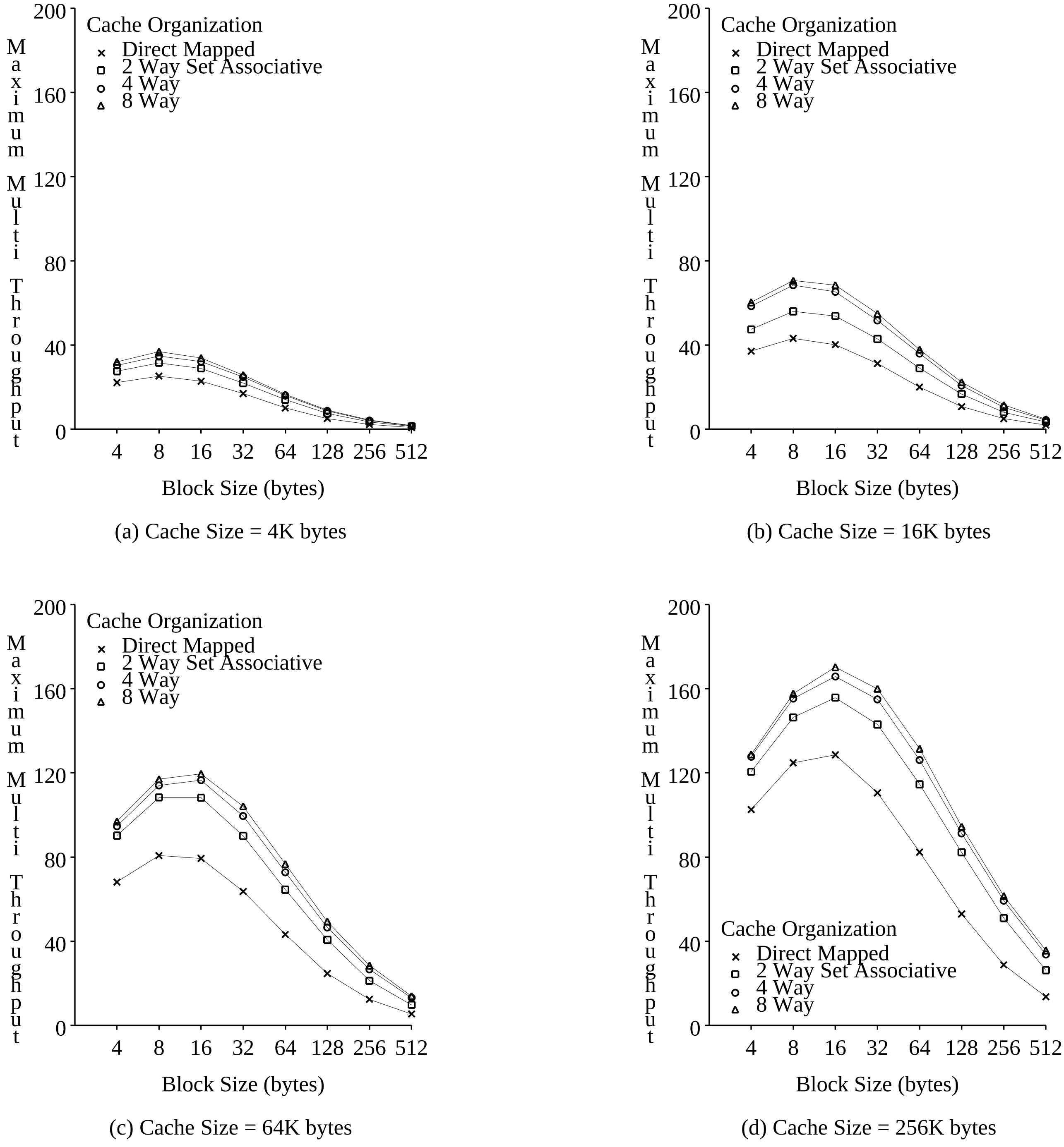


Figure 4.8: Maximum Multi Throughput (in VAX MIPS) with Varying Cache Set Associativity

in which T_m^C is independent of the cache set associativity, and a decrease in the value of $M \times T_m^C$ due to a decrease in the value of M alone may not be sufficient to offset the increase in T_C^P due to the increased cache set associativity. Furthermore, the lower per-processor bus utilization with an increased set associativity allows more processors to be connected together, and improves the multiprocessor throughput, even though the throughput of each processor might suffer. Keep in mind that caches in multiprocessors serve to reduce memory latency as well as to increase *system* throughput (by reducing the demand for the shared bus), whereas the main purpose of a cache in a uniprocessor is to reduce memory latency and to improve *uniprocessor* throughput. Therefore we see that a larger set associativity may be warranted in a multiprocessor even though it may not be warranted in a uniprocessor with similar memory system parameters.

Also observe that set associativity has little effect on the choice of the best block size. This reinforces the results of Section 4.2.2 on block size choice that were derived for direct mapped caches.

4.2.4. Bus Choice

From the results presented in Figures 4.4 and 4.5, it is clear that an STP bus can provide much better maximum system performance than a circuit switched bus, especially when the main memory latency is large. Furthermore, an STP bus is able to sustain maximum system performance for a wide range of main memory latencies. However, for low main memory latencies, circuit switched buses can compete in performance with STP buses.

Finally, the bus width choice is considered. Figure 4.9 shows the performance impact of increasing the bus width. In all cases the main memory latency is 250ns and an STP bus is used. For both 64K and 256K bytes caches, doubling the bus width from 4 bytes to 8 bytes increases the maximum multi throughput by about 50% (at block size of 16 bytes.) Each further doubling of the bus width improves performance less (about 30%). A wider bus decreases the block transfer time and consequently the bus utilization and the queuing delay. The reduced queuing delay and bus transfer time improve the read latency and the throughput of an individual processor; the reduced bus utilization allows more processors to be added to the system. Increasing the bus width appears to be an effective way of improving system performance, but has diminishing returns. It warrants investigation when a system is being designed, just as any other memory system parameter.

While the results on bus choice are not unexpected, it should be emphasized the we need to include all components of the memory system in evaluating any design choices and determining the magnitude of the maximum

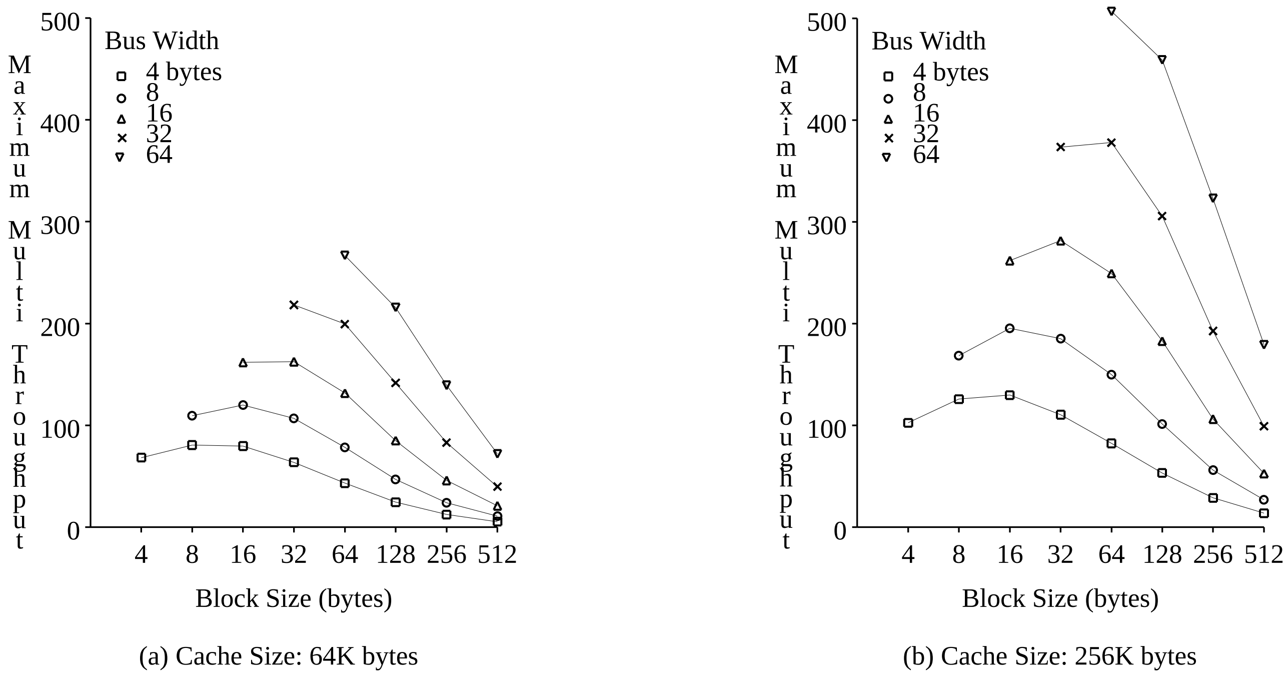


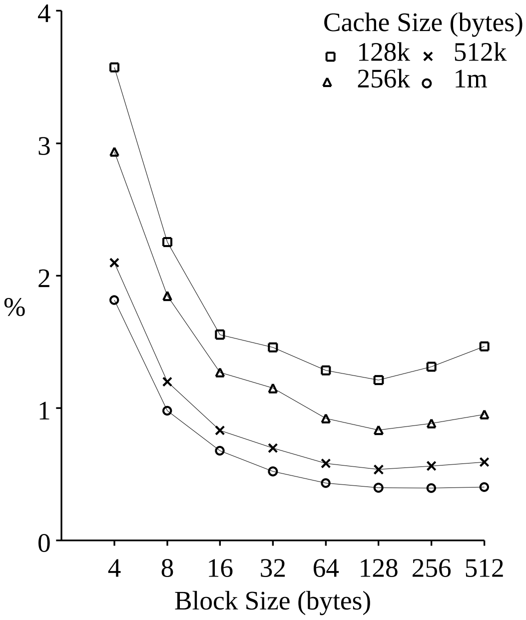
Figure 4.9: Maximum Multi Throughput (in VAX MIPS) for STP Bus with Varying Bus Width

system processing power. Furthermore, the analytical models allow one to determine quantitatively the magnitude of performance difference between arbitrary design choices in the memory system.

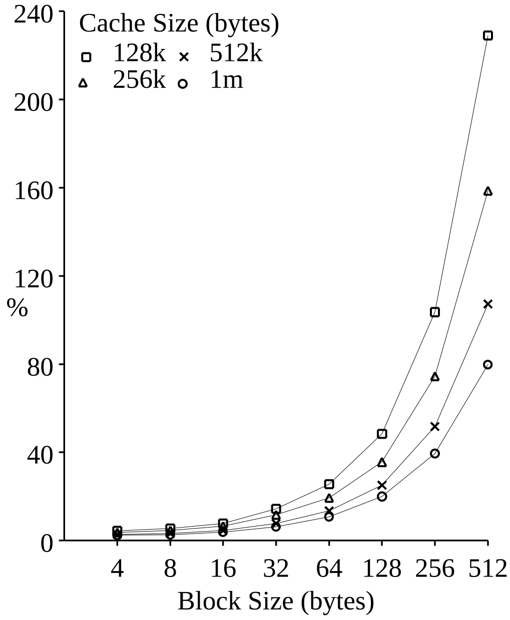
4.3. Results From TITAN Traces

4.3.1. Cache Performance Metrics

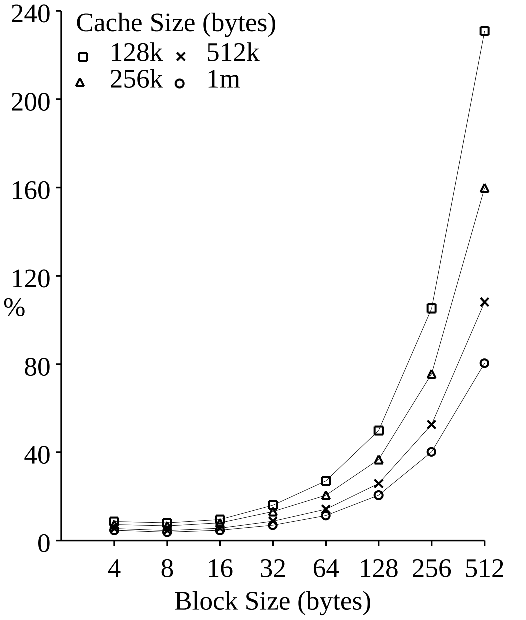
The miss ratio (in percentage) and bus traffic ratio (also in percentage) are presented in Figures 4.10(a)-(c), for different cache and block sizes. The results in Figure 4.10 are obtained under the same conditions as in Figure 4.1, except that the TITAN instead of ATUM traces and larger caches are used in Figure 4.10. The miss ratio and bus traffic ratio in Figure 4.10 are slightly lower than those derived from ATUM traces in Figure 4.1 for the common cache sizes of 128K and 256K bytes. With the TITAN traces the lowest miss ratio is 1.2% and 0.83% respectively for the two cache sizes at the block size of 128 bytes; whereas with the ATUM traces the miss ratio is 1.3% and 0.76% at the block size of 256 bytes. It should be noted in Figure 4.10 that the miss ratio continuously decreases significantly with increasing cache size. With 1M bytes cache the miss ratio can be reduced to as low as



(a) Miss Ratio



(b) Bus Traffic Ratio (Data Only)



(c) Bus Traffic Ratio (Data and Address)

Figure 4.10: Cache Performance Metrics for the TITAN Traces

0.4%. Although the absolute decrease of the miss ratio is small, if the throughput of a multiprocessor is to be maximized the performance improvement is more related to the relative decrease of the miss ratio, as been demonstrated from the results with the ATUM traces. Therefore using caches of 1M bytes or more is justified for a multiprocessor system constructed with Titan-like processors, a result that will be confirmed in the upcoming sections.

The best block size is 64 to 128 bytes if we only consider the miss ratio. But as shown in Figure 4.10(b)-(c), when the block size is larger than 32 bytes the bus traffic ratio becomes much larger than when the traffic ratio is the lowest (at the smallest block size of 4 bytes). Therefore neither miss ratio nor bus traffic can be used to guide the choice of the block size for a multi.

4.3.2. Uniprocessor Performance

The effect of varying main memory latency on the uniprocessor throughput is shown in Figure 4.11. The results are obtained under the same conditions as in Figure 4.2, except that different traces and different ranges of cache sizes are used. The two trends observed in Figure 4.2 are still apparent in Figure 4.11, i.e., the uniprocessor throughput suffers from the increase of the main memory latency, and suffers more when the block size is smaller. But difference also exists between the two figures. For example, in Figure 4.2(d) with the cache size of 256K bytes the uniprocessor throughput is decreased only moderately when a large block size such as 256 or 512 bytes instead of the best block size (32 or 64 bytes) is used. But in Figure 4.11(b) the performance is reduced sharply if such a large block size is used. The performance becomes less sensitive to the block size as the cache size is increased. But even when the cache size is as large as 1M bytes (see Figure 4.11(d)) the performance degradation is still more significant than in Figure 4.2(d).

The uniprocessor throughput in Figure 4.11(d) is more sensitive to the memory system design (such as the choice of the block size) than in Figure 4.2(d). However, the miss ratio in Figure 4.2(d) (derived from 256K bytes caches with the ATUM traces) is always higher than the miss ratio in Figure 4.11(d) (derived from 1M bytes caches with the Titan traces). This is because the performance degradation due to memory access actually depends on the *frequency of the access* to the main memory. The frequency of the access to the main memory is the product of the *miss ratio,* the *number of memory accesses per instruction,* and the *instruction execution rate of the processor*. The numbers of memory accesses per instruction for the two processor architectures are comparable (about 2 for the VAX processor, and 1.5 for the Titan processor). But the instruction execution rate of a Titan processor is about eight times higher than that of a VAX processor. So even though the miss ratio of the Titan processor with a 1M

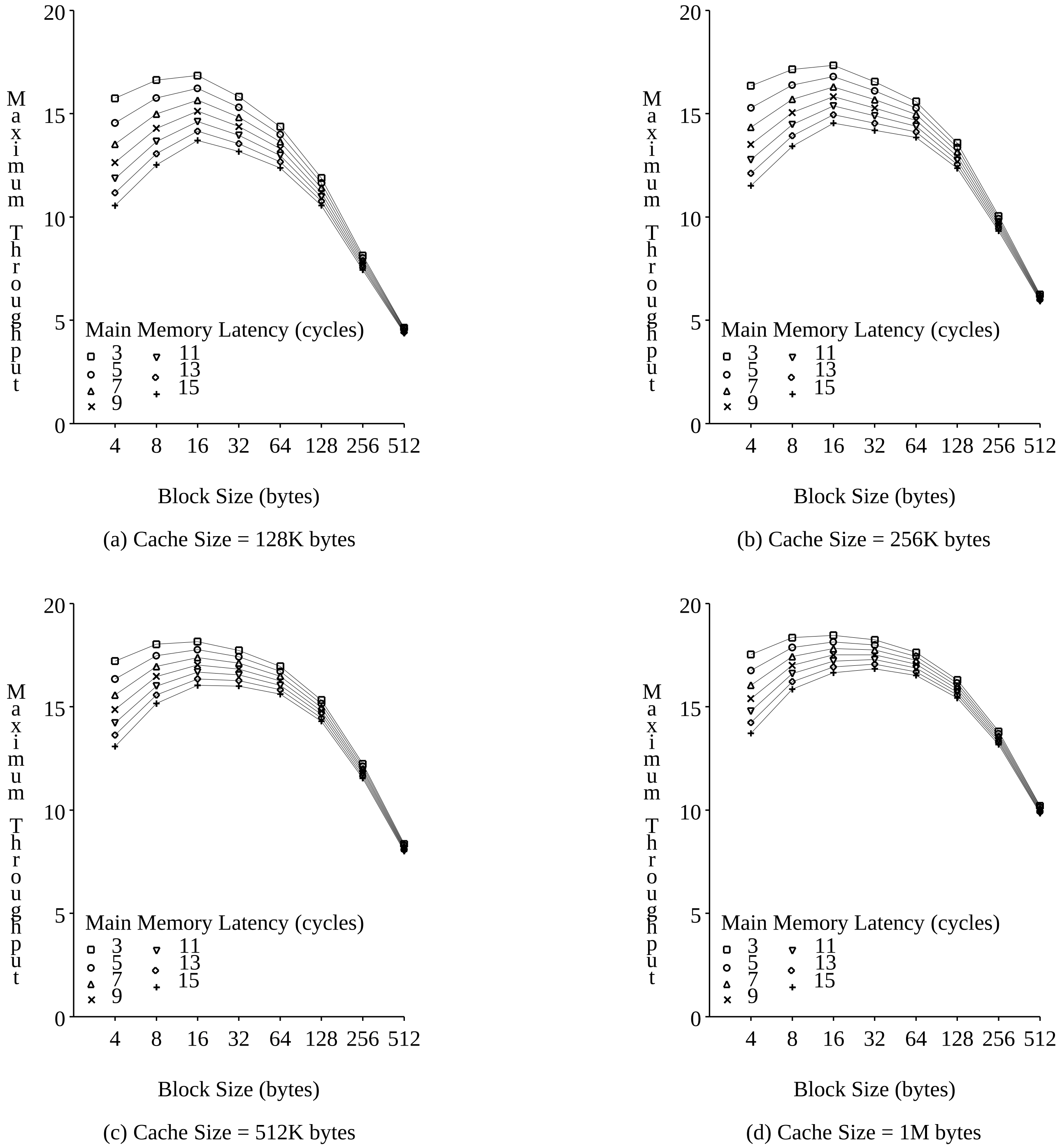


Figure 4.11: Uniprocessor Throughput (in Titan MIPS) with Varying Main Memory Latency

bytes cache is always lower the processor makes access to the main memory more often (in real time) than the VAX processor with a 256K bytes cache. Therefore averagely the Titan processor is blocked more often from instruction execution, and the performance degradation due to the increase of the memory access time (in this case, the cache miss latency increases with the block size) is higher than that of the VAX processor.

The relative speed of the Titan processor to the VAX processor, in fact, can also be used to explain another difference between Figure 4.2 and Figure 4.11. Comparing these two figures we can easily see that the performance of the Titan system is more sensitive to the change of the main memory latency.

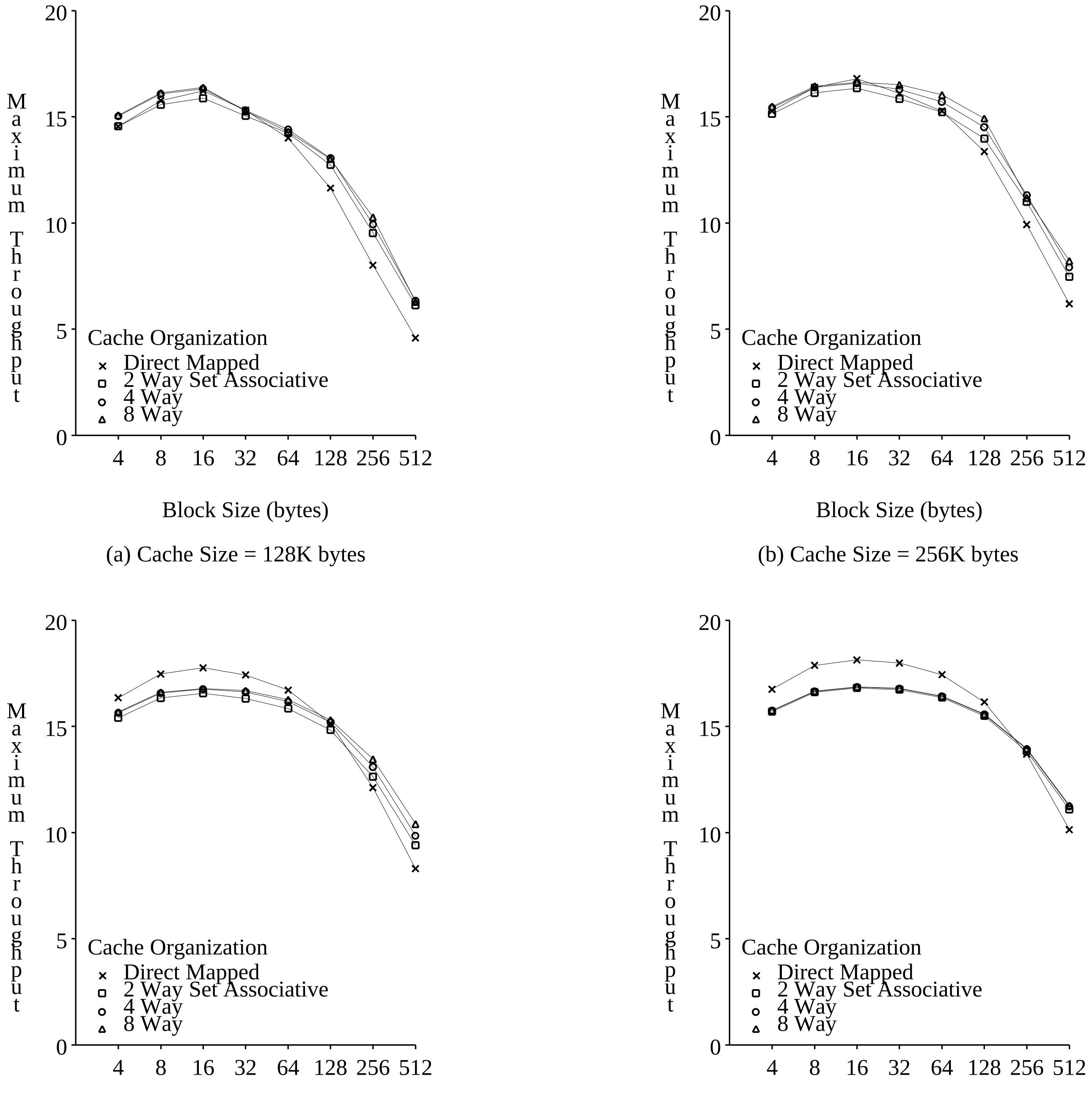
The effect of having different cache set associativity on uniprocessor throughput is shown on Figure 4.12. As in Figure 4.3, the main memory latency is fixed at 250ns and direct mapped caches are assumed to be 10% faster than the caches with set associativity of 2, 4, or 8. As in Figure 4.3 the need for set associativity is reduced with the larger caches, and a wider range of block sizes can be chosen. The major difference between Figure 4.12 and Figure 4.3 is that, with the TITAN traces the miss ratio advantage of a set associative cache over a direct mapped one can not compensate its cycle time disadvantage when the cache size reaches 512K bytes; whereas with the ATUM traces this happens when the cache size is 64K bytes or larger. This probably is because the problem size used to derive the TITAN traces is much larger, and set associativity can still be very effective in reducing the miss ratio when the cache size is 256K bytes or smaller.

4.3.3. Cache Block Size Choice

Figures 4.13(a)-(d) present the maximum multi throughput (in Titan MIPS) with various cache sizes and main memory latencies for a circuit switched bus, and Figures 4.14(a)-(d) present the same for an STP bus.

Certain observations made in Figure 4.4 and Figure 4.5 remain true here. For example, peak system performance with the circuit switched bus degrades severely when the main memory latency is increased. In Figure 4.13, where the circuit switched bus is used, the maximum multi throughput with 1M bytes caches and the best block size (16 bytes) decreases from 192 Titan MIPS to 122 Titan MIPS, when the main memory latency is increased from 150ns to 450ns. Peak system performance with the STP bus, on the other hand, is still quite insensitive to the change in the main memory latency, as depicted in Figure 4.14.

The choice of a cache block size as dictated by the results from TITAN traces, however, is not the same as when the ATUM traces are used. For example, using ATUM traces with the circuit switched bus the best block size



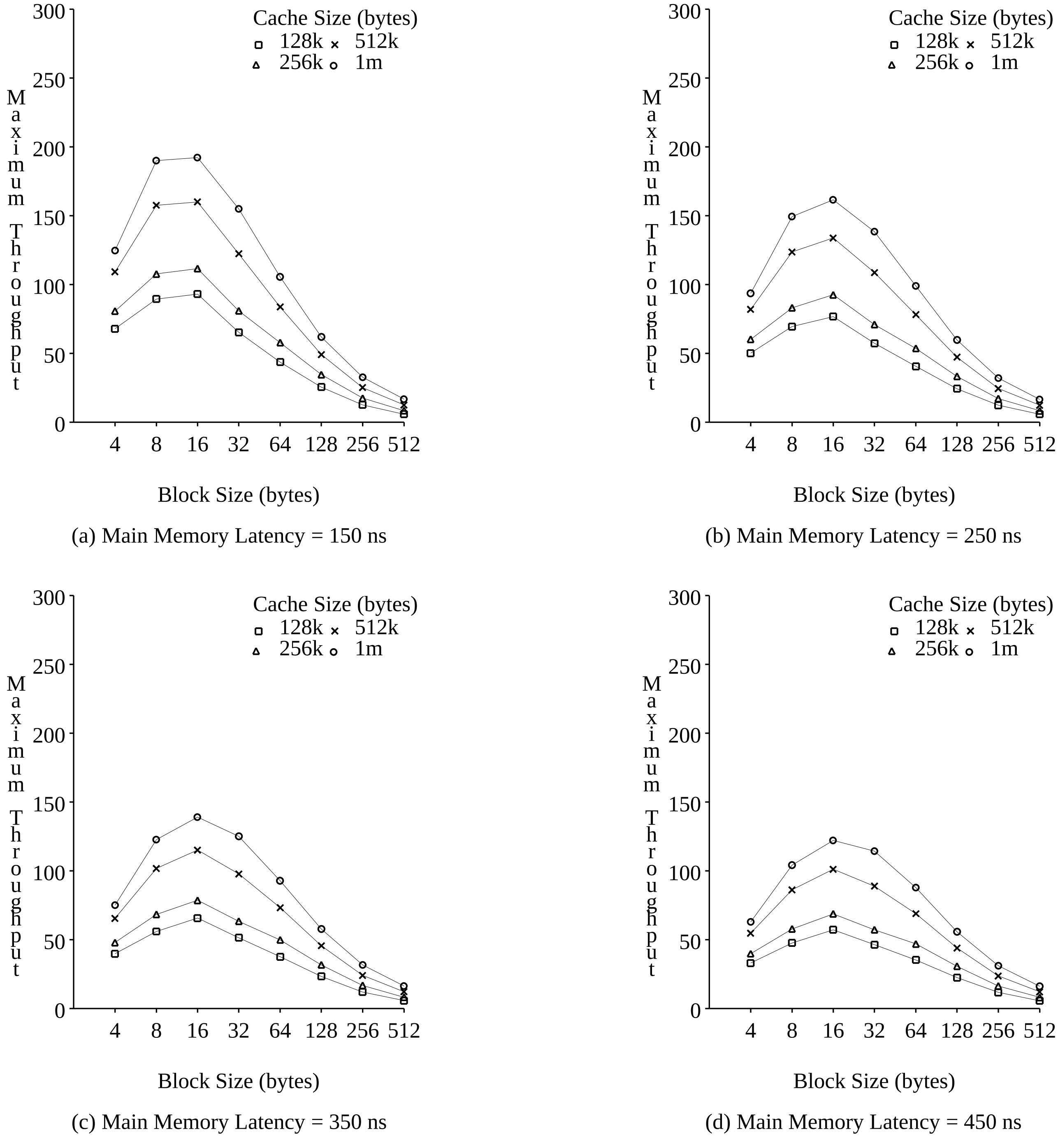
(a) Cache Size = 128K bytes

(b) Cache Size = 256K bytes

(c) Cache Size = 512K bytes

(d) Cache Size = 1M bytes

Figure 4.12: Uniprocessor Throughput (in Titan MIPS) with Varying Cache set Associativity



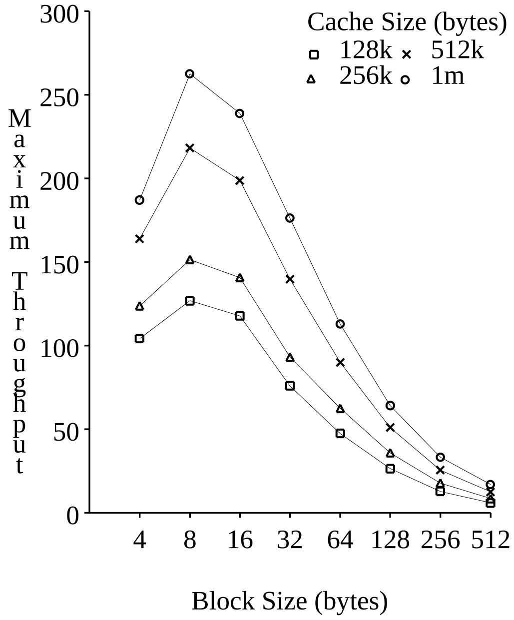
(a) Main Memory Latency = 150 ns

(b) Main Memory Latency = 250 ns

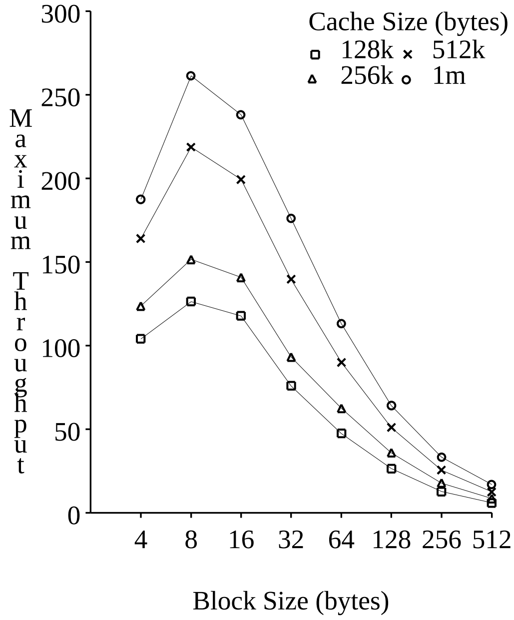
(c) Main Memory Latency = 350 ns

(d) Main Memory Latency = 450 ns

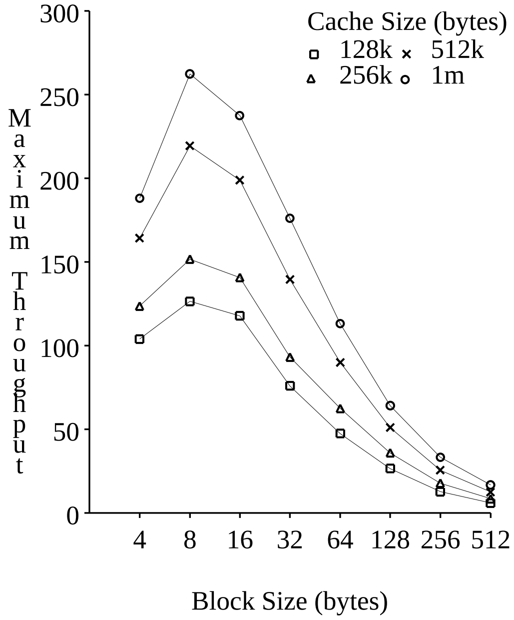
Figure 4.13: Maximum Multi Throughput (in Titan MIPS) with a Circuit Switched Bus



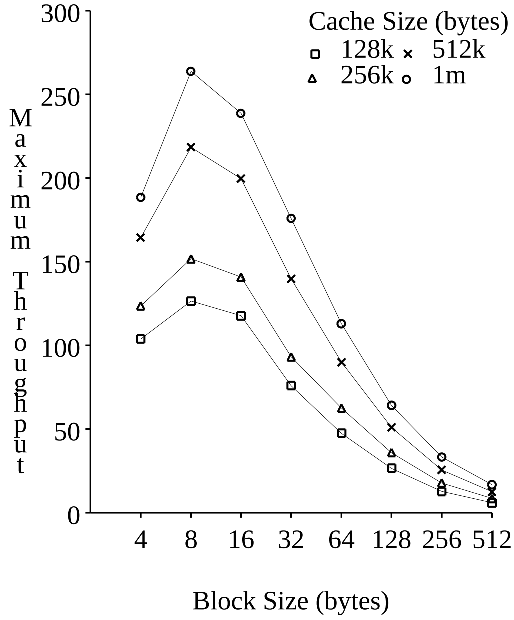
(a) Main Memory Latency = 150 ns



(b) Main Memory Latency = 250 ns



(c) Main Memory Latency = 350 ns



(d) Main Memory Latency = 450 ns

Figure 4.14: Maximum Multi Throughput (in Titan MIPS) with an STP Bus

for the 128K and 256K bytes caches gradually shifts from 16 bytes to 32 bytes when the main memory latency is increased from 150ns to 450ns. But the best block size calculated from the TITAN traces (16 bytes) does not change throughout the same range of the main memory latency for the two cache sizes. In fact the best block size remains the same even as the caches become as large as 1M bytes.

Cache size also does not affect the choice of the block size when an STP bus is used (see Figure 4.14). The best block size is always 8 bytes, smaller than when the circuit switched bus is used (16 bytes). However, the most important result, that the maximum multi throughput is insensitive to the changes of the main memory latency, is the same as derived from the ATUM traces.

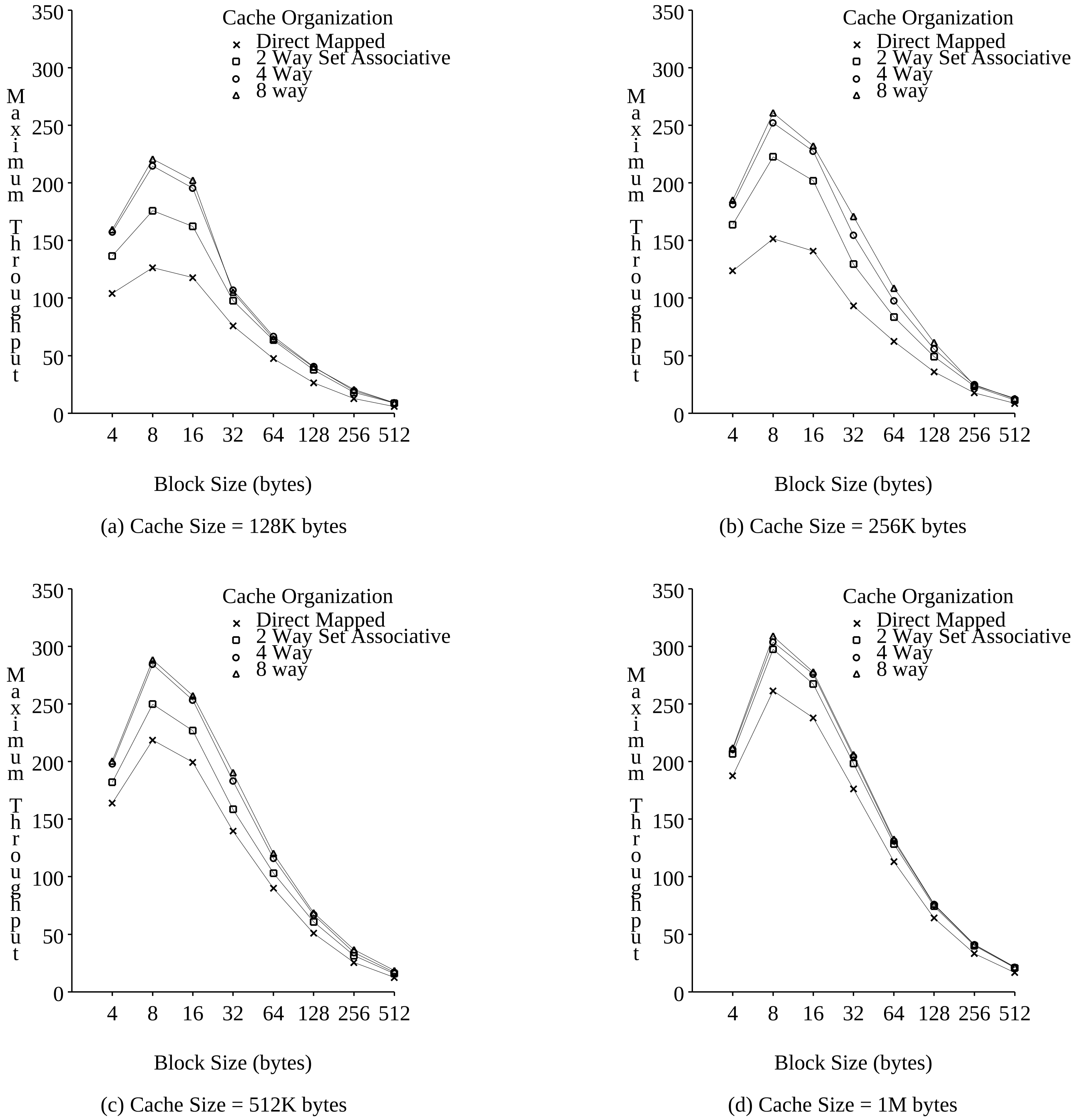
The performance improvement from the use of an STP bus instead of a circuit switched bus is more prominent with the TITAN traces. With 256K bytes caches and 150ns main memory latency, the maximum multi throughput (using the best block size) increases from 112 Titan MIPS to 151 Titan MIPS, an increase of 35%, when the circuit switched bus is replaced by an STP bus. For the same cache size and main memory latency the improvement on the VAX multi is 28%. Using 1M bytes caches the performance improvement on the Titan multi from the use of a better bus switching method is still an impressive 37%.

4.3.4. Cache Set Associativity Choice

The maximum multi throughput is improved by the use of set associative caches, as indicated by Figure 4.15. Figure 4.15 is obtained under the same conditions as in Figure 4.8, except that the TITAN traces are used in Figure 4.15. Figure 4.15 shows that, with the cache size as large as 1M bytes, set associativity can still contribute more than 14% of the performance increase. The performance gain is even larger when smaller caches are used. The maximum multi throughput is increased by 39% and 47% when 2-way set associative instead of direct mapped caches are used for 128K and 256K bytes cache respectively.

4.3.5. Bus Width Choice

Figure 4.16 shows the effect of increasing the bus width on the maximum multi throughput, assuming the main memory latency is 250ns and an STP bus is used (same conditions as in Figure 4.9). As apparent in Figure 4.16 increasing the bus width is the most effective way to increase the system performance. For example, using 8 bytes instead of 4 bytes wide bus, the maximum multi throughput (using the best block size) increases from 261 Titan MIPS to 360 Titan MIPS, or by about 39%. Further increase of the bus width still improves the maximum



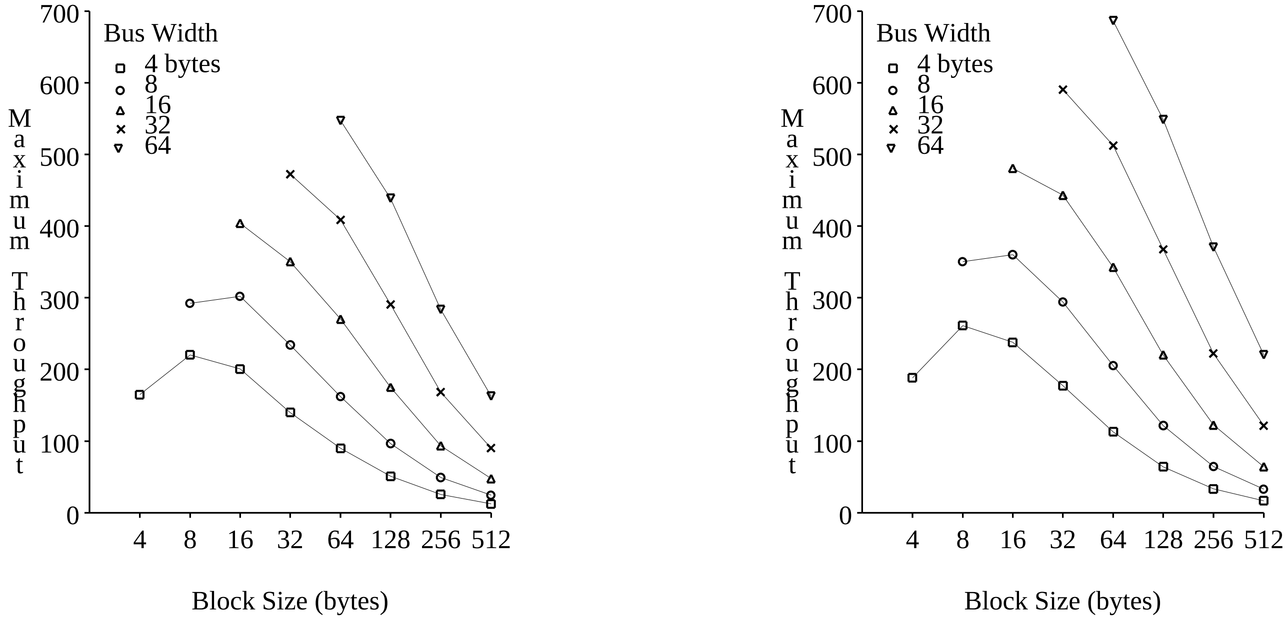
(a) Cache Size = 128K bytes

(b) Cache Size = 256K bytes

(c) Cache Size = 512K bytes

(d) Cache Size = 1M bytes

Figure 4.15: Maximum Multi Throughput (in Titan MIPS) with Varying Cache Set Associativity



(a) Cache Size: 512K bytes

(b) Cache Size: 1M bytes

Figure 4.16: Maximum Multi Throughput (in Titan MIPS) for STP Bus with Varying Bus Width

multi throughput, but to a lesser extent.

It is interesting to note that, as the bus is widened beyond 16 bytes the best block size, i.e., the block size that gives the highest multi throughput is always the same as the bus width. This is because using larger blocks (than the bus width) the decrease of the miss ratio is so small that it does not compensate the negative effect of increasing the bus bandwidth demand from each processor. For example, if a 32 bytes bus is used with 1M bytes caches, the miss ratio is reduced from 0.52% to 0.40%, a decrease of 23% when the block size is changed from 32 bytes to 512 bytes. But the amount of data traffic generated on bus is increased by 16 fold. Taking into account the bus bandwidth spent on the overhead address transfer, the bus bandwidth demand from each processor still increases by at least 10 times. This effectively reduces the number of processors that can be connected to the bus to less than one tenth.

4.4. Summary and Conclusions

In this chapter the design choices of the memory system for shared bus multiprocessors (multis) operating in a multi-user, throughput-oriented environment were evaluated. The evaluation was done using the CMVA models and processor memory reference characteristics derived from the widely-used ATUM and TITAN traces.

The results show that simple consideration of traditional uniprocessor performance metrics (such as miss ratio and bus traffic), independent of the parameters of the shared bus and the main memory, is likely to result in erroneous conclusions. With a circuit switched bus, it is especially important to consider all components of the memory system, including the main memory latency. With a split transaction, pipelined (STP) bus, main memory latency is less crucial to maximum multi throughput, but the best block size is neither the one that results in the lowest cache miss ratio nor the one that results in the lowest bus traffic. Also, an STP bus is preferable to a circuit switched bus if the system performance is to be maximized. The performance of an STP bus can be further improved, with diminishing returns, by increasing the bus width.

The block size that delivers the maximum multi throughput is 8 to 32 bytes with the ATUM traces, for the cache sizes ranging from 4K bytes to 256K bytes, and for different cache organizations and bus switching methods. With the TITAN traces the best block size is 8 or 16 bytes for the cache sizes between 128K and 1M bytes. The choice of the block size depends strongly on the bus width. We have assumed a 4 bytes wide bus in this performance evaluation. But as the bus technology evolves much wider bus is available and a larger block size is likely preferred. A larger block may also be desired if we assume that, after a cache miss the requested data in the missing block is always returned first, and the processor can resume execution as soon as the requested data arrives at the cache [10].

The need for set associativity in the caches was also considered. Although the importance of set associativity in uniprocessor caches diminishes when the cache size is as large as 1M bytes, set associativity was found to be desirable in multiprocessors even with such large caches. This is because the additional set associativity reduces the per-processor bus bandwidth demand and allows more processors to be connected together in the multi, thereby increasing the maximum multiprocessor throughput.

Chapter 5
Speedup Oriented Environment

5.1. Introduction

When a multiprocessor is operating in a speedup-oriented environment the workload consists of parallel programs. The major performance goal is to maximize the speedup of individual programs. To achieve this goal most of the design considerations of the multiprocessor still center around the same memory system components as those considered in the throughput-oriented environment. In other words, the design effort involves finding the suitable values for the different design parameters of caches, shared bus and main memory. A natural starting point is to find out the workload that characterizes the execution of parallel programs, and use the workload to investigate all interesting parameter values in the design space. As shown in the previous chapters where experiments are conducted for the throughput-oriented environment, the task is simply to pick the best parameter sets that achieve the design objectives.

However, the nature of parallel workloads is sufficiently different from the workload for the throughput-oriented environment that the straightforward approach used in the previous chapters is not applicable in the speedup oriented environment. For the parallel workload the most notable problem that compounds the effort of designing the memory system is that certain aspects of the workload may change drastically with the choices of the system components. For example, as will be shown later in this thesis, the number and the type of memory access generated by a parallel program can change with the number of processors, and the kind of hardware synchronization mechanism used in a multiprocessor. Therefore a more sophisticated methodology is needed to take into account this dependency between the workload and the memory system of the multiprocessor. This consideration becomes the major motivation of using an *execution driven* simulation method, where a running parallel program is maintained throughout the simulation to evaluate the performance of parallel execution.

But even with the execution driven simulation method the dependency between the parallel workload and the memory system of a multiprocessor is not totally taken into account. The performance of a parallel program depends not only on the hardware configuration of the multiprocessor that executes the program, but also the parallelizing strategy used in the program. Since the best parallelizing strategy may also be a function of the hardware characteristics of a multiprocessor, each parallel benchmark program must be separately parallelized and simulated for each multiprocessor configuration. For each different multiprocessor configurations simulation should be conducted with different parallelizing methods for each parallel benchmark program. This creates a very large design space with different software and hardware design parameters.

The very large design space considered, coupled with the use of the very time-consuming execution driven simulation method, greatly limits the area that can be explored in the design space. In particular, all the hardware design parameters, such as cache size, block size, associativity, etc., except the number of processors and the synchronization mechanism are fixed throughout the study, and only five parallel programs are included in the benchmark set. As a result in this thesis the two goals of the performance study in the speedup-oriented environment are: 1), to study the way the memory access pattern changes with the number of processors for the five benchmark programs; and 2), to evaluate the performance of some software/hardware synchronization mechanisms suited for bus-based shared memory multiprocessors. In the next section the design issues related to these two goals are elaborated.

5.2. Design Issues

The basic idea behind parallel processing is using *divide and conquer* to speedup the solution of a problem. But the strategy does not come without introducing additional costs. With multiple processors working on the same problem, coordination, or synchronization among these processors can add a substantial amount of overhead to the execution. Parallel programs also have quite different memory access patterns from that of their serial versions. At the least, the combined memory requirement from all processors can easily saturate some hardware resource such as the shared bus, of which the capacity does not scale proportional to the number of processors. All these factors contribute to the less than perfect speedup of parallel programs. These two overhead costs of parallel processing are elaborated below.

5.2.1. Synchronization Overhead

During the execution of a parallel program processors need to communicate with each other. Exchanging information may require processors to observe a certain prescribed order of memory access temporarily, i.e., making processors synchronize with each other. Enforcing the order of memory access increases the execution time in two ways. First, additional synchronization traffic is generated on the bus. Second, some processors may have to wait for others during a synchronization event. Many novel software and hardware ideas have been proposed to reduce either or both synchronization costs. For example, to reduce the synchronization traffic, *test&test&set* has been introduced to reduce the number of test&set memory accesses generated by waiting processors. The *Software Queue* method further cuts down the amount of unnecessary synchronization traffic [6, 58]. Reducing the processor waiting time during a synchronization event is more difficult, and may require the cooperation from the compiler and the operating system. A compiler can schedule a balanced workload to each processor, while the operation system can adjust the run time process scheduling policy to minimize the process waiting time [75]. Even the synchronization mechanism can be redesign to tolerate the difference in processor arrival times of a synchronization event [33]. Synchronization can also create a unique memory access problem called *hot-spot* [52]. The problem is effectively dealt with by the *combing* strategy [52, 63, 74].

5.2.2. Memory Access Pattern

Except for the added synchronization code, the total number of instructions executed and memory accesses made by a program should not change too much by parallelizing the program. However, distributing computational work among multiple processors can drastically change the memory access pattern. For one thing, the memory system now must handle the combined memory requirement from all processors, and the total memory demand can easily overwhelm the memory system. A hierarchical memory system traditionally used to reduce average memory access time for a uniprocessor, can serve the additional purpose of alleviating the memory access contention to the memory system.

The most common implementation of a hierarchical memory system takes advantage of private caches. Caches can reduce the memory demand to the shared bus and main memory. But introducing private caches to a multiprocessor also creates the *cache coherence* problem. Many cache coherence protocols have been proposed, and the trade-off between complexity and performance under different workload conditions analyzed. For bus based multiprocessors a particular simple and efficient solution employs the *snooping* strategy [26, 29, 38, 47, 50, 66].

Bus traffic induced by cache coherence events can be substantial. The exact amount of coherence traffic generated is sensitive to the type of snooping protocols used, and the sharing patterns of the parallel workload. A challenge to the design of either a parallel program or the memory system of a multiprocessor is to minimize the cache coherence traffic while the maximum speedup is pursued. This requires a thorough understanding of how the memory access pattern is changed when different number of processors are used.

5.3. Performance Evaluation Technique

5.3.1. Previous Methods

Traditional methods used to evaluate the performance of uniprocessor systems include hardware measurement, analytical modeling and trace-driven simulation, which may use real or pseudo traces. These methods can also be used to evaluate the performance of multiprocessors. For example, the performance of different software synchronization algorithms was measured on a Sequent Symmetry multiprocessor [32], and the performance of different cache organizations on different models of Symmetry multiprocessors [7]. The performance of different cache coherence protocols was studied using analytical modeling [69, 70], trace driven simulation with real traces [4, 18, 23, 24], or pseudo-traces [9]. Real parallel trace can either be derived from hardware monitoring [60], or software means such as the *ptrace* method. The parallel trace itself was analyzed to evaluate the performance of cache snooping protocols [22], or different design trades-off of the directory-based cache coherence scheme [72]. However, these methods are not always as effective as for uniprocessor systems. For example, the accuracy of these methods depends on the accuracy of the parallel traces, or the accurate characterization of the traces. Yet the trace or the trace characteristics of a parallel program may vary widely with the system configurations, in particular the memory system. This makes the trace or the trace characteristics derived from a specific multiprocessor configuration either unsuitable or of limited use for evaluating the performance of multiprocessors with different memory configurations.

5.3.2. Methodology Used in This Thesis

To overcome the above shortcomings, my methodology for evaluating the performance of parallel processing will be based on two ideas: **Execution-Driven Simulation** and **Study of Individual Computational Phases**. Below I will give a simple description and the rationale of the two ideas. More details can be found in the next two chapters.

5.3.2.1. Execution Driven Simulation

In order to obtain a more realistic simulation result the *execution-driven* simulation is used. That is, instead of decoupling trace generation and trace usage in multiprocessor simulation, these two phases are combined into a single stage of simulation. A software multiprocessor simulator, which integrates the functions of all relevant multiprocessor components including processors, private caches, shared bus and main memory systems, is built to simulate the execution of a parallel program. Therefore during simulation the *performance feedback* to the parallel execution is correctly taken into account, and accurate statistics can be derived.

5.3.2.2. Study of Individual Computational Phases

Since speedup is the most important concern in designing both parallel algorithms and multiprocessors, we are certainly interested in the execution time of the programs. But the interactions between parallel programs and memory system are so complex that simple speedup values may not provide enough information to guide the design process of either. An example is given in Figure 5.1.

Figure 5.1 shows the speedup of several parallel programs that have been run on my multiprocessor simulator. In Figure 5.1 the execution times of different programs are derived from the same set of multiprocessor

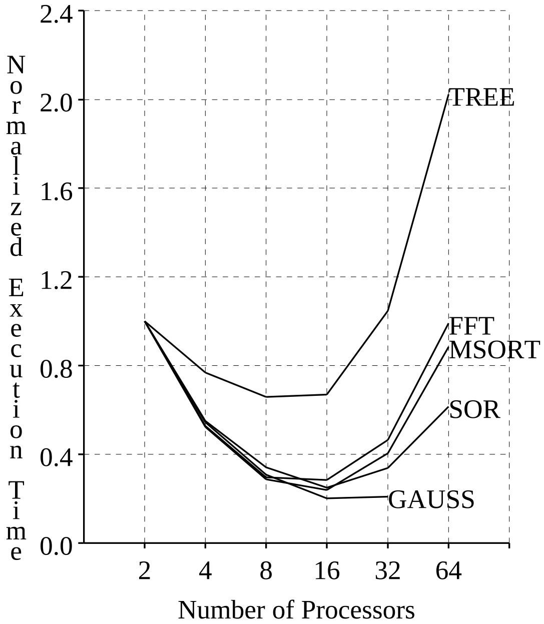


Figure 5.1: Normalized Execution Times of Some Parallel Programs

This figure shows the execution times of five different parallel programs. The execution times of each program are obtained from multiprocessor simulation, and are normalized to the execution time when the benchmark is run with two processors. Both the parallel benchmark programs and the simulation will be discussed in detail in later chapters.

configurations. The execution time of each program is normalized to the execution time when the program is running with two processors. The figure gives dramatically different pictures for different programs. Although analyzing the program code can offer some explanations to the speedup characteristics, we can not tell which one is more, or how much more, important than the others. Therefore a better way to characterize parallel program execution on a given multiprocessor is needed to answer these questions.

Some earlier research in this field has tried to examine some program execution statistics other than the total execution time. For example, the composition of the shared and non-shared access, read and write access, or the miss ratio of these different types of access have been analyzed. These statistics serve as an approximation of a certain component of performance. But the information are not enough in pin-pointing the exact cause of performance problems.

A more sophisticated approach has been taken in [72], which tried to link the invalidation traffic pattern to the behavior of high-level data objects in parallel programs. From the invalidation pattern the authors were able to suggest, for example, efficient ways to reduce invalidation traffic generated by synchronization objects. I intend to go a step further by finding other important and useful characteristics of parallel execution. To this end I will take a slightly different approach. Instead of linking a particular program behavior to high-level data objects, I try to correlate it with program *code segments*, i.e., different execution phase of a program.

Taking statistics separately for different program code segments has many advantages. My preliminary study has shown that, during the execution of a parallel program different parts of the code use hardware resource in very different ways. It will be easier to analyze the execution statistics separately for different parts of program code because the characteristics of the statistics should be simpler than the combined one. In this way the relationship between parallel programs and multiprocessors can be better understood. Analyzing parallel execution based on code segments may also be better than based on data objects. For example, some shared data may be written during the serial computation phase by one processor and then read during the parallel computation phase by all processors. This suggests write broadcasting, if restricted to only the serial computation phase, is not an expensive operation because only one processor will use the bus.

Chapter 6
Characteristics of Computational Phases

6.1. Introduction

In this chapter I will describe how the history of a complete execution of a parallel program is divided into different computational phases. A qualitative analysis of the memory access behavior for each computational phase will then be followed. The memory access pattern is strongly related to the cache coherence protocol used. In the following discussion an invalidation based cache coherence protocol is always assumed. The protocol is similar to *Illinois Protocol* and has been use by Symmetry machines [50]. The protocol employs four cache block states: *readonly-private, readonly-shared, dirty* and *invalid*. More detail about the protocol can be seen when the multiprocessor simulator is discussed in the next chapter.

6.1.1. Definitions

Before going further a few terms that may have different meanings in other places are defined.

Non-Shared Access: which can be instruction fetch or private data access.

Write Miss: write access that results in bus operation. Thus besides the normal cache miss, i.e., the data to be modified is not in the cache at all, write miss also includes write hit to a cache block that is in the *readonly-shared* state. The latter case requires an invalidation signal be broadcast on the bus.

Speedup Characteristics: the change of a certain performance measure when the number of processors is increased. The performance measure, for example, can be the execution time or the cache miss ratio. Note that the term does not imply the performance measure should increase or decrease with the number of processors.

P: number of processors.

N: problem size. This number refers to the size of the major data structures in a problem. For a linear array *N* is the number of elements in the array, and for a two-dimensional matrix, the number of elements in a column (or a row).

6.2. Division of Computational Phases

A parallel program code is divided into several disjoint segments so that each segment will have distinct execution characteristics. The most important characteristic is the bus demand, because the bus bandwidth is a major hardware resource likely to become a performance bottleneck. Another important characteristic is the synchronization behavior, since synchronization is unique in the way it utilizes hardware and software resources. Since different code segments are disjoint, at any time during execution only one code segment is active. In other words, at any time a running program is in a certain **computational phase** that is associated with a specific code segment.

Several different code segment groups have been identified and each of them is associated with one of the following computational phases: i) *Scheduling*, ii) *Independent Computing*, iii) *Communication*, iv) *Barrier*, v) *Barrier-Spin Wait*, and iv) *Serial*. A typical parallel program consisting of these different code segment groups is shown in Figure 6.1[1]. The definition and the characteristics of each computational phase will be given in the following sections.

6.2.1. Scheduling Phase

When a loop is parallelized there are two ways to distribute, or schedule the computation to processors. If *static scheduling* is used, a fixed set of loop iterations is assinged to each processor before execution of the loop. The alternative is *dynamical scheduling*, which comes in various complexity and costs [53, 64, 65]. Unless specified otherwise, the scheduling strategy used in this thesis is the *simple dynamic self-scheduling* method, by which each processor takes its turn to go through a scheduling critical section to obtain one iteration for independent

```
serial
loop {
    scheduling
    independent computing
    communication
}
barrier
barrier-spin wait
```

Figure 6.1: Code Segment Groups of A Parallel Program

[1] This example is more appropriate for the program with *DOALL* loops, where all iterations of the loop are ready to execute in parallel. My thesis study is restricted to programs with this type of parallel loops. Parallel loops may also be the *DOACROSS* type, where dependency relationship exists between loop iterations [20].

computation. The advantage of static scheduling is small scheduling overhead during run time, while dynamic scheduling offers better load balancing under different input conditions and uneven progress of processes created by the operating system scheduler [53]. The choice of a scheduling strategy can also affect the execution of other computational phases. The exact effect will be discussed when the relevant computational phases are encountered. Here only the scheduling overhead of dynamic scheduling is analyzed.

The execution time of the scheduling phase is mainly affected by the time to schedule one unit of work to a processor, and is usually done inside a critical section. If the critical section is implemented by *test&test&set*, a major concern is the saturation of the critical section. When this happens a waiting queue will form at the critical section and a large amount of bus traffic will be generated by the waiting processors. Theoretically the bus traffic generated from scheduling one processor can be as high as $O(p^2)$, p being the number of processors in the waiting queue. But with the FCFS bus arbitration policy, about $O(p)$ of bus accesses are likely to occur [28]. Whether a queue will form or not depends on the ratio of the time to schedule a unit of work and the time to process it. The scheduled work is processed in the independent computing and communication phases. The characteristics of the waiting queue will be further discussed when these two computational phases are analyzed.

6.2.2. Independent Computing Phase

During the independent computing phase a processor is actively doing computation without synchronizing with others. The independent computing phase is the only place where speedup is possible, i.e., the program execution time can be reduced when more processors are used. Since the total amount of computation does not usually change with the number of processors, doubling the number of processors cuts the amount of work in half for each processor. However, the same amount of computation does not mean other hardware requirement will not change. For example, if *bus demand* is defined to be the total number of the bus accesses (due to cache miss and invalidation) made by all processors, the bus demand during independent computational phase can increase, as will be shown later, with the number of processors.

Bus demand from the initial load of non-shared data, which includes instructions, data in private stack and heap, is proportional to the number of processors used, because every processor needs a copy of these data. By assuming the use of infinitely large cache and disallowing process migration, as will be the case of our simulation, initial load becomes the only bus traffic generated by non-shared data access. Initial load of non-shared data tends to encounter high bus access latency because all processors are likely to have their initial load at the same time.

However, this one time cost can be amortized and become insignificant if there is a large amount of computation in a program.

Bus demand from shared data access may also grow with the number of processors. Although the initial load of shared data causes cache miss as non-shared data does, long term bus demand results from data sharing, and does not diminish with time no matter how large the cache is. Three major factors account for the increase of the bus demand from shared data access when more processors are used: *fine grain data sharing*, *reference spreading*, and *scheduling policy*[1]. They will be discussed in detail in the next three sub-sections.

6.2.2.1. Fine Grain Data Sharing

When a shared data item has a fine grain sharing pattern it has very little *processor locality*. That is, the data is repeatedly read and modified by different processors, resulting in very high read and write miss ratios for the data. With loop level parallelization fine grain sharing can happen frequently because it is often difficult, or impossible, to schedule the same processor as both the writer and the reader of a data element for successive executions of a parallel loop.

Fine grain sharing usually occurs when a shared data structure is jointly modified by multiple processors. A commonly seen access pattern of the data structure is that *multiple* data elements of the shared data structure need to be read before any data element is updated. The updated data element may in turn become part of the reading lists when other data elements are modified. In other words the value of one data element depends on the values of multiple elements, which may in turn depend on the values of more elements. If all data elements in this data dependency relationship in successive executions of a parallel loop can not be confined to a sub-section of the data structure, there is no way to schedule one processor to be the sole reader and writer for any part of the shared data structure. When this happens each data element is bound to migrate from one processor to another during each execution of the parallel loop.

The shared read miss ratio of a shared data structure that is subjected to fine grain sharing can increase rapidly with the number of processors. For example, suppose an just updated data structure is evenly distributed among all

[1]There are still other, but much less frequently seen factors that can contribute to the same effect. For example, in the *asynchronous* or *chaotic* algorithms such as the one used in our benchmark SOR (detail of the benchmark can be found in the next chapter), the new value of each data element in a shared data structure is calculated from the old values of other data elements in the data structure. Since new values are *immediately* available as soon as they are calculated, old values will be used fewer number of times before they are replaced by new ones when more processors are used.

processors, i.e., each processor has an exclusive copy of an equal proportion of the data structure. The next time when the data structure is read in a parallel loop, the miss ratio of *initial loading a random part* of the data structure will be $(1 - \frac{1}{P})$, since there is only a $\frac{1}{P}$ chance for a processor to find the data in its local cache. With only a few processors this read miss ratio can easily come close to 100%. The actual miss ratio for the data structure, however, can be lower since each data element of the data structure can be re-used by the processor during the execution of the parallel loop. But even so, using more processors reduces the chance that a loaded data be used again by the same processor.

Fine grain sharing can also increase the shared write miss ratio. A shared data structure is usually read and modified alternately during successive parallel computations. An element of the data structure will more likely be in multiple caches if it is read in multiple iterations when more processors are used. So when the data is updated in the next parallel computation the write miss ratio, which as defined earlier includes invalidations from write hits to the shared copies, can easily reach 100% even when only a few processors are used.

6.2.2.2. Reference Spreading

Reference spreading is related to shared read accesses [44]. When a loop is parallelized, each processor executes some iterations of the loop. Certain shared data, mainly *loop invariants*, may be read in all or some iterations before the data is updated again. Since each processor takes fewer loop iterations to work with when more processors are used, loop invariants will be read fewer times by the processor. All the read accesses except for the first time are cache hits. Since the total number of accesses to loop invariants is independent of the number of processors, and all except the owner of the loop invariants (usually the processor that does the computing during the serial phase) need to load them, the read miss ratio of the loop invariants is proportional to $(P - 1)$.

6.2.2.3. Scheduling Policy

Bus demand of shared data access can also be affected by the scheduling policy. For certain parallel programs it is possible to use the *static* scheduling method, i.e., assigning a fix subset of a shared data structure to each processor to avoid fine grain sharing of the data structure. The shared data do not migrate at all, resulting in very low miss ratio.

When the alternative *dynamic scheduling* method is used, a processor may take a different part of the data structure during different executions of the same parallel loop. For a shared data structure that is constantly read and modified different parts of the data structure are likely to migrate from one processor to another, resulting in a very high miss ratio. Dynamic scheduling may even increase the miss ratio of a data structure that is *readonly* or *mostly readonly*. Since a random part of the data structure is read each time a parallel loop is executed, a processor may take a different part of the data structure during different executions of the parallel loop. Therefore the processor is likely to load a larger part of the data structure than when static scheduling is used.

The difference in the cache miss ratio caused by the use of different scheduling policies can be very large when fine grain sharing can be totally avoided by static scheduling. With static scheduling access to the ''resident'' data set generates cache miss only during the initial load. So the long term miss ratio can be very low. If dynamic scheduling is used, every time the same parallel loop is revisited a processor has approximately a $\frac{1}{P}$ chance of working on the same data set as the last time. So the miss ratio of the initial load of the data during each execution of the parallel loop becomes proportional to $(1 - \frac{1}{P})$, or $\frac{(P-1)}{P}$, just as if fine grain sharing has happened to the data structure. Even with a moderate number of processors the miss ratio becomes quite close to 100%.

The advantage of longer cache data residency with a statically scheduled loop can disappear, however, if loop execution is interspersed with dynamically scheduled loops that read/write the same data structure. In this case the cache miss ratio of the statically scheduled loop will be similar to that when dynamic scheduling is used.

6.2.3. Communication Phase

For some programs there may exist a communication phase during the execution of a loop iteration. The communication phase usually comes immediatedly after an independent computing phase. During the communication phase a processor reads/writes some shared data to exchange information with other processors. The exchange is conducted within a critical section, and participated by some or all processors. The execution time of the communication phase can be strongly affected by the contention to the critical section. Contention becomes higher when more processors share the same critical section (multiple communication critical sections may exist for different sub-groups of processors). Contention also varies depending on the scheduling policy. With static scheduling contention can be high because processors are more likely to reach the communication critical section at the same time. On the other hand, if the loop is dynamically scheduled and the size of the communication critical section is small, the contention problem can be much less severe. This is because processors enter the independent computing phase at different times, and are less likely to reach the communication phase at about the same time.

6.2.4. Barrier Phase

As the last operation of a parallel loop, barrier serves as a rendezvous point for processors. The barrier operation is split into two computational phases: *barrier* and *barrier-spin wait*. During the barrier phase a processor registers its arrival, usually by incrementing or decrementing a barrier counter atomically. The atomic increment/decrement operation can be implemented in hardware, as in the case of Symmetry. The alternative is the much less efficient way of decrementing the counter inside a critical section, which generates much more bus traffic and delay.

When barrier is implemented by an atomic increment/decrement instruction it generates one bus access for each processor. The duration of this instruction is so short that no queue is likely to develop for the access to the barrier counter, or if it ever forms because very large number of processors are used, will disappear very quickly. Therefore the major cost of the barrier operation is one bus access time, a very small fraction of the total execution time.

One important characteristics of the barrier operation is that usually the number of times a processor executes barrier operations grows much slower than the the total amount of computation in a program, as the problem size increases. So when the problem size is very large, the time spent in the barrier phase becomes even less significant.

6.2.5. Barrier-Spin Wait Phase

After going through a barrier phase a processor enters a barrier-spin wait phase, spin-waiting for the end of the barrier operation[2]. The execution time of the barrier-spin wait phase consists of three parts: 1) waiting time for other processors to finish their independent computation, 2) waiting time for other processors to pass the barrier phase, and 3) memory access time to read the end of barrier signal.

The second and the third time components, though proportional to the number of processors, are usually much smaller than the first component. The first component is difficult to quantify because it is affected both by the parallel algorithm and the input data values. But even if the barrier-spin wait time is highly variable, its speedup characteristics shows a certain trend that is related to the scheduling policy of the parallel loops.

---

[2]The alternative to spin-waiting is either to relinquish the processor (be swapped out by the operating system) if the processor utilization should be maximized. But processor utilization is not a major concern here. The interested reader, however, can find a thorough study of the topic in [75].

When static scheduling is used the execution time of the first time component is affected by the difference in the execution times of different iterations in the loop. When fewer processors are used each processor takes more units of work to compute in the independent computing phase. Since the *variance in execution time* of two units of work is always larger than that of one unit [3], the fewer processors used, the more units of work each processor takes, and the larger the time difference between processors to finish their independent computing phase. In other words the more processors are used, the smaller this spin-waiting time component.

If dynamic scheduling is used after the independent computing phase a processor has to go through the scheduling critical section for the last time (to detect the end of loop) before executing the barrier operation. So the barrier-spin wait time is at least equal to the total time for the scheduling critical section to schedule all the processors. If the number of processors is small processors are not likely to reach the scheduling critical section at the same time, and the time to reach the barrier is determined by the time a processor finishes the independent computing phase. The barrier-spin wait time is determined by the difference of processor execution times in the independent computing phase. The situation is similar to the case when static scheduling is used, and the barrier-spin wait time will decrease when more processors are added.

But when more and more processors are used processors start to accumulate at the scheduling critical section. In this case the different arrival times at the barrier will more likely be a result of the ‘‘staggering’’ effect of scheduling, i.e., processors leave the scheduling critical section one by one, with the interval determined by the scheduling rate. The time between the first processor leaving the scheduling critical section to enter the barrier phase, and the last one doing the same, is equal to the time for the scheduling critical section to schedule all processors. So the time for each processor to spend in the barrier-spin wait phase begins to increase with the number of processors.

The increase in time due to the increase of processors can be dramatic if the number of processors is large enough that a long queue has formed in the scheduling critical section. Since the total time to schedule all processors increases more than linearly with the number of processors, if a long queue forms in the scheduling critical section (see Section 6.2.1), the increase of the execution time of the barrier-spin wait phase is also faster than linearly with the number of processors.

[3] $\mathrm{Var}\,[X + Y] = \mathrm{Var}\,[X] + \mathrm{Var}\,[Y]$, if X and Y are independent random variables [68].

6.2.6. Serial Phase

Serial code can be executed in two ways. The first uses one processor to compute while others spin wait. The second way allows all processors to do the same computation, even though every processor expects to get the same result. The reason that we may want to use all processors to compute the same thing is that the overhead of using only one processor can be avoided. This overhead includes a test to determine which processor should do the computation at the beginning of a serial phase, and a bus access by every processor to detect the end of the serial phase. The disadvantage of making all processors compute the same thing is that some data may have to be loaded to all instead of only one cache. Since processors tend to make bus access *at about the same time*, the bus access time of these data can become proportional to the number of processors used. Therefore this type of serial computation should be used only when the amount of serial computation is very small.

Most of the time serial code is executed by one processor. In our parallel programs this type of serial computation is always preceded by a *barrier* operation. Since those (all but one) processors that do not participate in the serial computing spin wait at the end of serial code after they decrement the barrier counter, their barrier-spin wait time can not be distinguished from the serial-spin wait time. So the the barrier-spin wait time is merged into the serial execution time. Therefore some earlier remarks about the barrier-spin wait phase can be applied to the serial phase.

The execution time of serial code can increase with the number of processors even when only one processor is engaged in computation. There are several reasons for this phenomenon. First, as shown earlier the barrier-spin wait time component may increase with the number of processors. Second, all processors need to make one bus access to detect the end of the serial phase. Since processors make this bus access at about the same time, the *bus access time is proportional to the number of processors used*. If the serial phase is short, this bus access time can become a significant part of the serial time.

The last reason for the increase in execution time is due to data sharing. If a shared data structure is jointly updated during a parallel computation, and if the data structure is read in a subsequent serial phase, a larger part of the data structure has to be loaded into the processor that does the serial computation when more processors are used. More precisely, the number of cache misses can be proportional to $(1 - \frac{1}{P})$, or $\frac{(P-1)}{P}$, since only $\frac{1}{P}$ of the data structure will initially be in the processor that does the serial computation.

Therefore the total number of cache misses R generated by *all* processors in the serial phase can be calculated by the following equation:

$$R = C_1 + C_2P + C_3\,\frac{P-1}{P}$$

where P is the number of processors and C_1, C_2, and C_3 constants. The second term C_2P represents the cache misses generated, for example, by the final broadcast in the serial phase. The third term $C_3\,\frac{P-1}{P}$ results from the kind of data sharing mentioned in the previous paragraph. The third term can reach its peak when only a few processors are used. The constant term of C_1 accounts for the rest of cache misses during the serial phase. Thus the above equation becomes close to a linear function of P when, for example, more than four processors are used.

If the barrier-spin wait time component is small, which is the case most of the time, the speedup characteristics of the serial phase is largely determined by the above cache miss equation. When a small number of processors is used the execution time is nearly constant because either the first term of the above equation dominates, or the bus utilization is so low that bus access time is not the major part of the execution time. When more processors are used the bus access time from the second term becomes more significant. Since cache misses in the second term tend to occur at the same time, the *bus access time becomes proportional to the number of processors*. Therefore if the actual serial computation is relatively short (hence the C_1 in the above equation is also relatively small), when enough processors (e.g., four) are used the execution time of the serial phase becomes a linear function of the number of processors used.

Chapter 7
Simulation Methodology and Benchmarks Used

7.1. Base Simulator

The multiprocessor simulation is conducted by a software multiprocessor simulator that simulates program execution in a Symmetry-like multiprocessor. The main feature of the simulator is that it is driven not by a memory trace as in the traditional trace-driven simulation, but by a running parallel program. Trace-driven simulation, which is used extensively to measure the performance of uni-processor systems, uses a set of memory traces as input to simulate the function of the memory system. The validity of the method is based on the assumption that the performance of the memory system does not change the instruction stream, hence the memory reference stream of a program. This assumption usually does not hold for the execution of parallel programs. A change in the performance of a memory system component may result in a change in the relative speed of processors, affecting the number and the interleaving of the memory accesses in the memory access stream. Therefore the multiprocessor simulator is designed so that it actually executes a parallel program during simulation. Memory traces are generated on the fly to drive the memory system, which in turn controls the progress of execution in each processor.

The multiprocessor simulator simulates the functions and the timing characteristics of all major components in a multiprocessor. During a simulation run, the simulator processes all memory access and synchronization events as soon as they are generated, and calculates the exact timings of these events. Thus at the end the performance statistics of the program and the usage of different system resources can be obtained.

The parallel benchmark program whose execution is to be simulated is compiled into the Intel 80386 machine language using the Sequent Symmetry C compiler. The multiprocessor simulator then interprets the program using the *ptrace* facility of Dynix. Because a ptrace call involves multiple context switches the major disadvantage of our method is the long simulation time. Simulation of a complete program execution is possible only when the problem size is small.

During a multiprocessor simulation memory access and synchronization events are generated according to the instruction interpreted. A memory access from a processor is first checked with its private cache, which may in turn generate a number of bus accesses. Bus accesses from different caches are queued for the shared bus. A bus operation may trigger additional activity in other caches or main memory, which may in turn generate new bus accesses.

A detail description of the function of different multiprocessor components in the simulator is given as follows.

7.1.1. Cache

The private cache uses an ownership based, invalidation type of cache coherence protocol that is similar to the Illinois protocol [50], and the one used in Symmetry. A cache block can be in one of the four states: *invalid, readonly-private, readonly-shared* and *dirty*. A cache block is in the readonly-private state when the block is clean and no other cache has the data, in the readonly-shared state when the block is clean and other caches may have clean copies of the data, and in the dirty state when the block is modified and is not written back to main memory. Each cache has two sets of tag memory, with one used for *bus snooping* and the other for *processor access*. The tag memory used for snooping control contains all four states to guarantee the proper function of the cache coherence protocol. The processor side tag memory has only three states (*invalid, readonly* and *dirty*), and is used to increase the cache availability to the processor. Since the processor side tag memory does not contain the complete information of a cache block, the processor sometimes has to check the bus side tag memory to decide what action to take for a memory access.

The snoop controller of each private cache monitors bus activity continuously. In response to certain bus events the controller changes the state of cache blocks on the bus side tag memory, and if necessary, on the processor side tag memory at a later time.

7.1.2. Shared Bus

The shared bus is a *split transaction,* or *packet switched* bus similar to the one used on Symmetry. Higher priority is given to reply requests (the responses to earlier read or read-modify requests), and the FCFS policy is used to resolve the remaining bus conflicts.

7.1.3. Main Memory

The main memory is modeled as a pipelined device which can accept one request on each cycle. This is realistic because requests can only come from the bus, which supplies at most one request on each cycle. The main memory latency is assumed to be constant.

7.1.4. Synchronization

Synchronization is done by *locking the private cache*. In the Intel 80386 architecture, a lock signal is sent from a processor when a *locked* instruction is executed[1] . During the execution of the locked instruction the local cache is unavailable to external bus requests. All the external bus requests to the locked cache will be queued and not processed until the locked instruction is finished. A locked cache does not forbid the transfer of ownership, however, even for the cache block that contains the lock variable. Therefore when multiple processors execute locked instructions that access the same lock variable at about the same time, the ownership of the block will be passed from one cache to another on each bus cycle. All the requests to the lock variable thus form a chain (or a queue) starting from the cache that currently has the lock. The chain is unwound later when the actual transfer of the cache block takes place.

7.2. Benchmarks

Five benchmarks are chosen for performance analysis. They are i) *GAUSS*, or Gaussian Elimination, which solves a linear system **Ax = B**, ii) *FFT*, or Fast Fourier Transform, which calculates the harmonic content of a motion, iii) *SOR*, which uses an asynchronous parallel successive overrelaxation method to solve a symmetric linear complementarity problem, vi) *MSORT*, which sorts an array of numbers, and v) *TREE*, which finds an arbitrary spanning tree of a graph. All programs are written in C and use the parallel library functions provided by Dynix. These programs are compiled into Intel 80386 machine language code before running by the multiprocessor simulator. The program code with comments is included in the Appendix.

7.2.1. Parallelizing Methods

Various parallelizing methods are used in these benchmarks. The simplest one makes a straightforward conversion from the serial code: distributing all iterations of the innermost (FFT) or the second innermost (GAUSS, SOR) loops to processors. Loop iterations of these loops can be executed in any order and little or no synchronization is needed between iterations. Therefore the amount of parallelism is proportional to the number of iterations in the loops, or equivalently, to the problem size. One feature of this parallelizing method is that, if parallelizing over-

---

[1]Any instruction in the Intel 80386 processor can become a locked instruction as long as the instruction is prefixed with a *lock* byte in the program code. Normally only the instructions that make access to lock variables are prefixed with the lock byte. The only locked instruction used in the five benchmarks is *xchgb*, or exchange-byte, a variant of test&set instruction.

head such as scheduling, communication (through critical section) and barrier operations is excluded, the total amount of computation is the same for the serial and the parallel algorithms.

The parallel algorithms used for the remaining two benchmarks (MSORT and TREE) are quite different from their serial counterparts. The innermost loops of the serial algorithms are not suitable for parallelization using the above simple method, because data dependencies exist between iterations of the loops. The parallel algorithms used for the two benchmarks are also quite different from each other, but they are all capable of creating $O(N)$, N being the problem size, of parallelism. The major drawback of these algorithms is that the total computation is increased by $O(\log N)$ from their serial versions. Because of the increase of computation these parallel algorithms have limited utility because potential speedup over the serial solution is realizable only when the number of processors used is comparable to the problem size. However, the emphasis in my study is to analyze the interactions between parallel programs and the multiprocessor that runs them, not finding the maximum speedup for a particular problem. Therefore the impracticability or inefficiency of parallel algorithms relative to their serial counterparts is not an important concern here.

7.2.2. Description of Program Structure

Since the speedup characteristics of a parallel program is strongly affected by the way the program is parallelized, I will describe each benchmark in detail, especially the way the major computation is partitioned and scheduled. I will also show program structures to facilitate the discussion of the simulation result. The program structures are described in pseudo-code as follows.

Parallel Loop (I,IM): a parallelized loop. The loop is *statically* scheduled, and thus consists of an **I**ndependent computing phase, or an **I**ndependent computing and a co**M**munication phase.

Parallel Loop (SI,SIM): also a parallelized loop. The loop is *dynamically* scheduled, and consists of a **S**cheduling and an **I**ndependent computing phase, or in addition to the two phases, a co**M**munication phase.

Serial: consisting of a barrier and a serial phase.

Barrier: consisting of a barrier and a barrier-spin wait phases.

As an example, a segment of parallel program in Figure 7.1, which is the innermost loop of the benchmark SOR, can be described in pseudo-code in Figure 7.2.

```
 1. while (1) {
 2.       GET_WORK(mywork, n)    /* scheduling critical section,
                                             fetching a loop index */
 3.       if (ixsi[mywork]) continue;
 4.                                  /* working on the ith column;
 5.                                         compute the scalar product of M  and x */
 6.       for (ww = q[mywork], j = ja[mywork];
 7.                                 j < ja[mywork+1]; j++)
 8.            ww += as[j] * x[ia[j]];
 9.       wt = x[mywork] - ome * diag[mywork] * ww;
10.       if (wt < 0.0) {
11.          ix[mywork]++;
12.          wt = 0.0;
13.       }
14.       xx = x[mywork];
15.       x[mywork] = wt;
16.       xx = dabs((xx - x[mywork]));
17.       S_LOCK(&difflock);     /* communication critical section; see
                                          whether further relaxation is needed */
18.            if (xx > diff) diff = xx;
19.       S_UNLOCK(&difflock);
20.  }
21.  my_S_WAIT_BARRIER2(bp, workp); /* barrier operation */
```

Figure 7.1: The Innermost Loop of The Benchmark SOR

In this figure `GET_WORK(mywork, n)` is a macro that implements simple self-scheduling inside a critical section; `S_LOCK(&difflock)` and `S_UNLOCK(&difflock)` implement the prolog and epilog of a critical section using the test&test&set code sequence; `my_S_WAIT_BARRIER2(bp, workp)` implements the barrier operation. All these macros are written in 386 assembly code. Line 2 represents the *scheduling* phase, line 3 to line16 the *Independent Computing* phase, line 17 to line 19 the *communication* phase, and line 21 the *barrier* and *barrier-spin wait* phase (the two phases are distinguishable in the assembly code extension of the macro).

```
Parallel Loop (SIM)              /* line 1 to line 20 of Figure 7.1 */
Barrier                          /* line 21 of Figure 7.1 */
```

Figure 7.2: Psuedo-Code Representation of the Program in Figure 7.1

7.2.3. GAUSS

Program Structure

```
loop {
    Parallel Loop (I)     /* calculate the ''weight''
                                    for each row to select pivot */
    Serial                /* select the pivot row */
    Parallel Loop (SI)    /* update each row */
    Barrier
}
Parallel Loop (I)         /* calculate the result */
Barrier
```

GAUSS, or Gaussian Elimination solves the linear system **Ax = B**, where **A** is a square matrix of order *N*, and **b** an *N*-vector. The matrix is transformed to an *upper-triangular* matrix with all diagonal entries nonzero by *forward elimination*. A subsequent *backward substitution* derives the solution for **x**. In the parallel algorithm the forward elimination step is modified so that a diagonal instead of an upper triangular matrix is created at the end. This modification makes it easier to distribute computational work evenly to processors. The pivot strategy used in the algorithm is *scaled partial pivoting* [19].

The complete computation involves multiple sweeps of the matrix. Each sweep starts with a parallel phase, in which a *weight* array is calculated from the matrix **A** for the latter pivot selection. This is followed by a serial phase, in which one processor selects the pivot row, and a parallel phase, in which all processors update the matrix independently based on the value of the pivot row. At the end of each sweep a barrier operation synchronizes all processors. After *N* sweeps through the matrix a parallel phase produces the final results.

Both static and dynamic scheduling methods are used in this program. Static scheduling is used in the parallel loop that calculates the weight array and the loop that calculate the final results. Only the parallel loop that update the matrix uses dynamic scheduling.

7.2.4. FFT

Program Structure

```
Serial
loop {
    Serial                /* two arrays exchange roles;
                                    calculate some loop invariants */
    Parallel Loop (SI)    /* transformation of array */
    Barrier
}
```

The input to the FFT (Fast Fourier Transform) problem is a motion in the time domain, and the output is the frequency content of the motion [19]. The input as well as the output is an array of numbers. The data in the input array undergoes multiple steps of transformation to generate the output at the end. In the program two arrays are used to carry out each step of the transformation: one stores the input array, the other the temporary result. At successive steps of transformation the two arrays exchanges roles as input and output arrays.

During each step of transformation a new array is generated. Since each array element can be independently generated, computation of one array element, which involves reading some elements of the input array and writing one element to the output array, is used as the scheduling unit. All computation is distributed to processors using a dynamic scheduling policy.

7.2.5. SOR

Program Structure

```
loop {
    loop {
        Serial                  /* calculate loop invariants */
        Parallel Loop (SIM)     /* update the array z */
        Barrier
    }
    Serial              /* check to see if further relaxation is necessary */
}
```

The SOR (Successive Over-Relaxation) method is used to solve the symmetric linear complementarity problem

$$M z + q \geq 0, \;\; z \geq 0, \; z\,(M z + q) = 0$$

A particular version of *asynchronous* SOR is used for analysis [42]. Matrix M is a read-only sparse matrix. Elements of the matrix are compressed and stored in an array, with separate arrays to store the row and column indices. Data elements in the array z are dynamically selected by each processor and updated, and the new values computed by processors are made available immediately to all other processors as soon as they are generated.

The structure of the parallel program is similar to that of FFT. The only difference is that at the end of the independent computing phase there is an additional synchronization operation in SOR. The operation involves a modification to a shared flag inside a critical section, to signify the intention for further relaxation or the end of computation.

7.2.6. MSORT

Program Structure

```
loop {
    Serial                  /* two arrays exchanges the input/output roles */
    Parallel Loop (SI)      /* merge sort */
    Barrier
}
```

MSORT sorts a list of numbers by using a combination of quick sort and merge sort. The list is first divided evenly into 64 parts. The sublists are sorted separately using quick sort. The sorted sublists are then put in a central task queue, where sublists are paired up and merged to form larger sublists. Merging sublists is done in parallel. Each process takes an element from one list and calculates its position in the merged list. The position of the element is calculated by a binary search in the other sublist to be merged with.

Although both quick sort and merge sort in the MSORT benchmark can be parallelized, the quick sort is less interesting and is accomplished with only one processor. Therefore performance statistics is collected only during the merge sort phase.

7.2.7. TREE

Program Structure

```
Serial
  Parallel Loop (I)         /* create the one-vertex forest */
  loop {
      Serial                /* initialize loop invariants */
      Parallel Loop (I)     /* short-cutting (doubling) to
                                 reduce tree height */
      Barrier
      Parallel Loop (I)     /* if a hooking is possible,
                                 determine where to hook */
      Barrier
      Parallel Loop (SI)    /* actual hooking of two trees */
      Barrier
  }
```

TREE derives an unrooted spanning tree from an undirected connected graph by using a modification of the Shiloach-Vishkin connected components algorithm [59]. Finding an arbitrary or qualified spanning tree from a graph is a common step in many graph algorithms. The serial algorithm visits each node and edge in turn and takes O(N) time. The parallel algorithm can process all the nodes and edges at the same time, but it requires only O(log$N$)

steps to finish.

The parallel algorithm tries to build up ‘‘connectivity’’ trees step by step, starting from a forest in which each tree contains just one vertex. At each step trees that are connected in the original graph are ‘‘hooked’’, or merged in pairs. Successive merging steps increase the tree size exponentially. Thus in O(log$N$) steps groups of connected vertices will form the largest possible tree, which immediately give the information of the connectivity of all vertices. The desired spanning trees are generated as a by-product of developing the connectivity trees.

During each step a processor repeatedly takes a vertex or an edge to operate. Both static and dynamic scheduling policies are used. The dynamic scheduling policy is used only when the processing time for a vertex or an edge is highly variable. The major data structures are arrays of records, which store the attributes of the nodes and edges. The topology of the trees and graphs, however, are represented by adjacency lists.

7.3. Simulation Conditions

The performance of a parallel program can be affected by many factors such as parallelizing strategy, data structures and multiprocessor configuration. Complete knowledge of speedup characteristics can be obtained by exploring all possible combinations of parameter values, a task that may require an unrealistic amount of time. Therefore in this simulation study the values of all multiprocessor system parameters except the number of processors are fixed. The research is then focused on finding most, if not all, of the important phenomena that characterize the speedup of parallel programs, instead of identifying all possible occasions that each particular may occur.

7.3.1. Multiprocessor Configuration

The values of multiprocessor parameters used in simulations are chosen so that they are as realistic as possible. Most of the values are close to the hardware specifications in the Symmetry multiprocessor. In particular, the multiprocessor has i) the zero-memory-wait-state processor speed of 3 cycles per instruction; ii) infinitely large, write-back caches with 4 byte cache blocks; iii) a split transaction bus with bus width of 4 bytes and multiplexed address and data lines; iv) main memory latencies of 3 cycles; v) 2, 4, 8, 16, 32 or 64 processors. Note that the minimum cache block size of 4 bytes is used to avoid the *false sharing* problem,[1] because it is hard to isolate and quantify its effect.

[1]When two or more processors are actively accessing and modifying disjoint sets of data, which happen to be allocated to the same cache block. This results in thrashing of the cache blocks among the competing processors.

When simulation starts each processor executes instructions continuously until a memory access is required. Checking the processor-side cache directory takes one cycle (details of the hardware organization can be found in Section 7.1). If the memory access is a cache hit the processor resumes execution immediately. Otherwise either the bus-side cache directory is checked, or a bus access is needed to put a request (which may be an invalidation) on the bus. Checking the directory or sending a request on the bus takes one cycle, in addition to the undeterministic queuing delay for the cache directory or the bus. Transferring a cache block (4 bytes) on the bus takes one bus cycle, and loading it into the cache takes another cycle. After the cache block is loaded the waiting processor re-issues the memory access, and spends one cycles for the expected cache hit. Write back of a cache block takes two cycles on the bus, one for the request (including address), and the other for the data.

7.3.2. Benchmark Program Size

The problem size of each benchmark has been kept small because of the slow instruction interpretation rate of the simulator. This deficiency is ameliorated by running each benchmark with two different sizes. By observing the trend of the result from the small to the larger problem size, the results for much larger problem sizes can be extrapolated.

The problem size limits the amount of parallelism available, which in turn determines the maximum number of processors that can be utilized. The matrix size in *GAUSS* is 32×32 or 64×64, in *SOR* 64×64 or 256×256. The array size in *FFT, MSORT* is 128 or 512, and in *TREE* 64 or 256. The data elements in all shared data structures are either integers or real numbers of 4 bytes.

Another important aspect of the benchmark programs is the executable code generated by the compiler. One major concern is the code it generates for critical sections. The current compiler on Symmetry generates a code sequence that implements *test&test&set*. Since the simulator interprets the code generated by the compiler, the basic synchronization mechanism used in the simulation is also test&test&set.

7.3.3. Conducting Simulation

Although the execution of each benchmark is simulated completely, statistics are not collected during the initialization part of the execution. Only after all processes are created and the program starts to solve the problem does the statistics collection begin. All processes are manipulated to start at the same time in the simulation, even though they are created, or forked at different times. Statistics are collected until the simulation stops, usually when the solution is generated in the output data structures.

Since the speedup characteristics of different parallel loops, especially the dynamically scheduled ones, can be very different even when all these loops belong to the same program, performance statistics have to be collected separately for each parallel loop so that the collected results are easier to analyze. Two benchmarks have multiple parallel loops: GAUSS and TREE. In each of the two benchmarks only the most important parallel loop is dynamically scheduled, with all the remainings statically scheduled (see the program structures in Section 7.2). Although statistics are collected for all the loops, in the following chapters where the collected results are analyzed, only the statistics of the dynamically scheduled loops will be discussed.

Statistics for shared access are collected separately. Shared accesses in the simulation are identified by their addresses, because on the Symmetry when a parallel program is compiled and lined, a particular memory region can be set aside exclusively for allocating shared variables.

7.4. Overview of Results

7.4.1. Simulation Statistics

Several simulation statistics are shown in Table 7.1 for the two-processor multiprocessor configuration. Table 7.1 lists the simulation time (simulated, or virtual time) and the average number of accesses, shared accesses and

Benchmark	Problem Size (array/matrix)	Simulation time (cycle)	Number of (per Processor)		
			Access	Shared Acc.	Instruction
GAUSS	32×32	1.8M	650K	110K	370K
	64×64	13M	4700K	744K	2700K
FFT	128	0.33M	110K	16K	70K
	512	1.7M	550K	82K	360K
SOR	64×64	0.5M	190K	32K	93K
	256×256	6.2M	2400K	410K	1100K
MSORT	128	0.3M	90K	10K	66K
	512	1.5M	460K	48K	340K
TREE	64	0.16M	58K	13K	31K
	256	0.64M	230K	50K	120k

Table 7.1: Simulation Statistics For 2-Processor Configuration

instructions made by each processor for each benchmark. The two benchmarks GAUSS and SOR, which use the larger data sets (matrices), execute more instructions and make more memory accesses. However, no matter how much different the number of accesses each benchmark makes, shared accesses are always about 11% to 23% of the total accesses, which includes instruction fetches. This ratio in fact is also quite insensitive to the change in problem size.

Simulation statistics are not shown for multiprocessor configurations with more processors. However, most of this information can be inferred from the discussion of the simulation results in the following chapters. In general, any of the above quantities can increase of decrease when more processors are used, depending upon the characteristics of the individual benchmarks.

The real time to conduct the simulation, however, always increases with the number of processors in a configuration. This is because the simulation, though a multi-tasking job by itself, essentially processes one event at a time in the simulated multiprocessor. Since the more processors used for a benchmark program, the higher the total number of bus accesses made from all processors, (an important finding in our study), the simulator needs to process more events and hence requires a longer simulation time. The simulation can interpret on the order of a few million instructions per day.

7.4.2. Organization of Result Discussion

Since the objective of this research is to demonstrate the distinct execution characteristics of different computational phases, the simulation results are discussed separately for each phase in the next two chapters (Chapter 8 and Chapter 9). Chapter 8 discusses the three most important and also close related computational phases in parallel execution: scheduling, independent computing and communication phases. Chapter 9 deals with the others, including the barrier, barrier-spin wait, and serial phases, for which I will demonstrate that their effect on parallel execution is very limited for our study environment. One major result from the discussion in these two chapters is that synchronization can be a performance bottleneck. This becomes the motivation for a separate chapter (Chapter 10) that is dedicated to the study of both hardware and software solutions to the synchronization problem.

Chapter 8
Scheduling, Independent Computing and Communication Phases

8.1. Introduction

In this chapter the program behavior in the scheduling, independent computing and communication phases of parallel program execution is discussed. The three computational phases are important because a parallel program spends most of its time in them. Various performance statistics such as the execution times and miss ratios in each phase are presented and analyzed. Special emphasis is put on the speedup characteristics, or the trend of performance statistics when more processors are used.

A separate section is dedicated for each of the three phases. In each section the speedup characteristics of the *execution time* leads the discussion. For easier comparison between benchmarks it would be better to present the execution times of all benchmarks in the same graph. But this is difficult because large disparities exist between the execution times of different benchmarks. Since comparing the times in their absolute terms is not a major concern in my study, they are normalized according to the benchmark used. The normalizing base chosen for each benchmark is the execution time of the benchmark when two processors are used. A *normalized execution time* is the true execution time divided by its normalizing base. Since the true execution time is seldom useful, unless specified otherwise, from this point the *execution time* simply means the normalized execution time, unless mentioned otherwise.

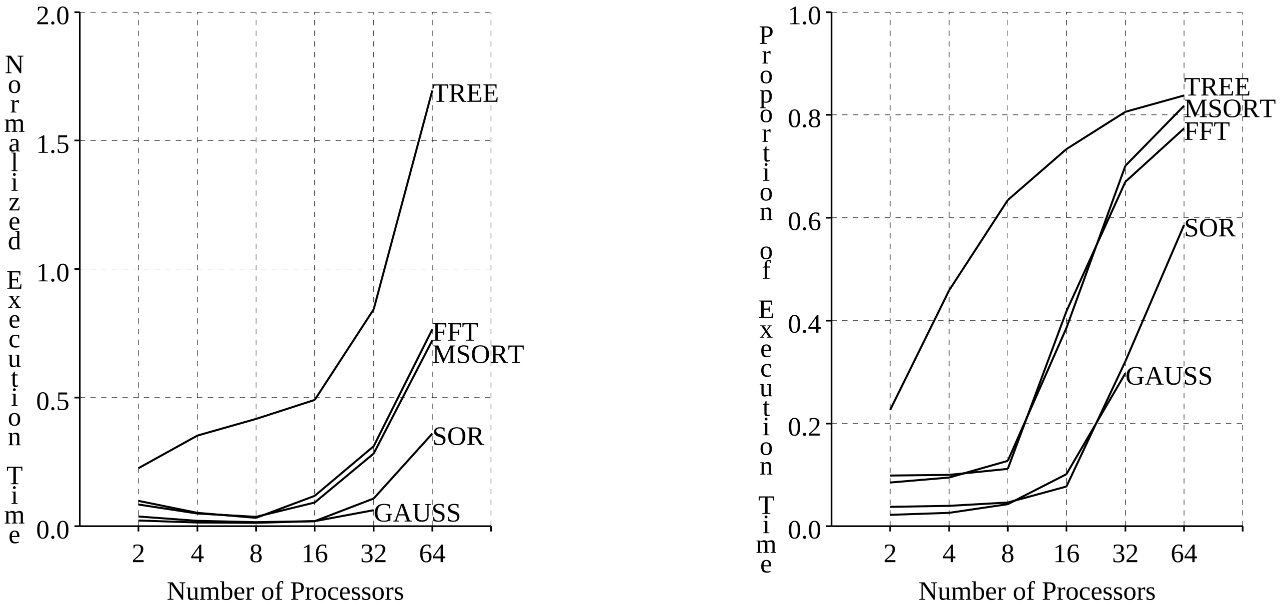
Note that another good choice for a normalizing base is the execution times of the *serial* solutions to the benchmark problems. In so doing the normalized execution time will immediately reflect the speedup of a parallel solution relative to its serial one. However, comparing the performance between the serial and the parallel solutions of a problem can be a very different research concern since the algorithm and coding style of a serial program may be very different from that of its parallel version. This study, though interesting by itself, falls outside the scope of this thesis research.

Since the multiprocessor simulation is slow, it is barely feasible to run with small problem sizes. To ameliorate the problem each benchmark is run with two different problem sizes. I will first give a detail analysis for the results derived from the smaller problem size, and then examine the results for the larger problem size. The objective is to find the trend in performance statistics so that I can project the result for the larger problem sizes.

8.2. Scheduling Phase

The normalized execution time of the scheduling phase, and the proportion of the execution time to the total time for each benchmark are shown respectively in Figures 8.1(a) and 8.1(b). Figure 8.1(a) indicates that all, except the execution times for the benchmark TREE, exhibit an initial decrease and a subsequent increase as more processors are added to the system. For the benchmark FFT and MSORT the scheduling time starts the upward swing (or at its lowest value) when 8 processors are used, and for the benchmark GAUSS and SOR, at the 16 processor

---



(a) Normalized Time

(b) Proportion of Time

Figure 8.1: Normalized Execution Times of the Scheduling Phase and the Proportion in Total Execution Times

Figure 8.1(a) shows the normalized execution time of the scheduling phase. The normalizing factor for each line (benchmark) is the total execution time of the benchmark when two processors are used. Each data point represents the scheduling time of only one parallel loop, which is dynamically scheduled with simple self-scheduling policy (see Section 6.1.3). All except the benchmark TREE show an a initial decrease in execution time when more processors are added to the two processor configuration. But eventually the execution times of all benchmarks start to increase when enough processors have been used. Figure 8.1(b) shows the proportion of scheduling time to the total execution time for each benchmark. In all cases the proportion of scheduling time becomes larger when more processors are used.

---

configuration. After this lowest point the scheduling time increases rapidly when more processors are used. For all except the benchmark TREE, doubling the number of processors can increase the scheduling time by more than 2-fold.

The scheduling time drops initially because each processor goes through the scheduling critical section fewer number of times as more processors participate in the computation. The scheduling time eventually starts to increase because the bus access latency becomes larger and the bus demand higher. The bus access latency increases because the bus utilization is higher as more processors are competing for the bus. The bus demand increases because if the scheduling critical section can not schedule processors fast enough, extra bus traffic will be generated by processors waiting to enter the critical section (explained in Section 6.2.1). Table 8.1 shows the increase in total bus demand by the shared accesses from all processors, the average number of processors inside or waiting to enter the scheduling critical section, and the average number of processors that are in the independent computing or communication phase.

It is clear from Table 8.1 that when the scheduling critical section starts or is about to saturate the bus demand can increase very fast (see highlighted entries for the benchmarks GAUSS, FFT & SOR, and the entries just above the highlighted entries for the benchmarks MSORT and TREE). In these entries (or multiprocessor configurations) the average scheduling queue lengths are at least 2.79 processors. Since processors waiting to enter the critical section can generate a lot of bus traffic (the amount of bus traffic generated by scheduling one loop iteration is proportional to the queue length. See Section 6.2.1) if the critical section is implemented with test&test&set [28], it is not surprising to see in Table 8.1 and Figure 8.1 that for each benchmark the surge in total bus demand coincides with the configuration when the scheduling critical section starts, or is about to saturate, and also coincides with a relatively large, if not the largest increase in the execution time when the number of processors is increased.

Table 8.1 suggests that, with loop level parallelization fewer than 32 processors can be used before the scheduling critical section becomes saturated. Using more processors will not reduce the execution time of a parallel loop whether the bus utilization has reached 100% or not, as indicated by Table 8.2. It is apparent from Table 8.2 and Table 8.1 that when the scheduling critical section saturates the bus may not have been fully utilized. Since the only two queues in the system are the scheduling critical section and the shared bus,[1] and only the shared

[1]Benchmark SOR has another shared resource: the communication critical section. But the critical section has smaller effect than the other two exclusive resources because processors spend only a small amount of time in this phase (see Figure 8.5 and Section 8.4).

Number of Processors	GAUSS			FFT			SOR		
	IB	QL	IC	IB	QL	IC	IB	QL	IC
2	-	0.09	1.91	-	0.20	1.8	-	0.08	1.92
4	1.3	0.22	3.78	1.1	0.42	3.58	1.5	0.17	3.83
8	1.8	0.75	7.25	1.2	1.02	6.98	1.4	0.46	7.54
16	2.6	3.74	12.26	**6.5**	**7.92**	**8.08**	1.9	1.82	14.18
32	**3.3**	**18.31**	**13.69**	2.4	25.44	6.56	**5.8**	**13.93**	**18.07**
64	-	-	-	2.4	58.68	5.32	3.5	46.24	17.76

Number of Processors	MSORT			TREE		
	IB	QL	IC	IB	QL	IC
2	-	0.17	1.83	-	0.87	1.13
4	1.2	0.39	3.61	3.0	2.79	1.21
8	1.4	1.08	6.92	**2.4**	**6.68**	**1.32**
16	4.1	6.82	9.18	1.8	14.51	1.49
32	**2.9**	**25.00**	**7**	1.8	30.47	1.53
64	2.4	58.66	5.34	2.1	62.39	1.61

IB: Increase of the Total Bus Demand;

QL: Average Scheduling Queue Length;

IC: Number of Processors in independent Computing or Communication phase;

Table 8.1: Increase of the Total Bus Demand from Shared Access, the the Number of Processors in Different Computational Phases

The total bus demand of shared access in the scheduling phase is the total number of shared misses (including invalidations) from all processors during this phase. The *increase in the total bus demand* of a particular configuration is the ratio of the total bus demand of the configuration to the total bus demand of the configuration that has only half as many processors. For example, for the benchmark GAUSS the total bus demand from shared access when the number of processors 32 is 3.33 times of the total bus demand when the number of processors is 16. The average *scheduling queue length* is calculated based on the proportion of time a processor spends in each of the scheduling, independent computing and communication phases. Since during the execution of a parallel loop a processor should be in one of the three phases, the average number of processors that can simultaneously be in any phase is therefore proportional to the time a processor spends in that phase. The average number of processors that can simultaneously be in the scheduling phase is also the *average queue length of the scheduling critical section*. The *number of processors that can simultaneously be in independent computing or communication phases* is calculated in the same way as the average scheduling queue length.

The entries (configurations) where the scheduling critical section starts to saturate (i.e., with 100% utilization) are highlighted (in boldface). In these entries the scheduling queue lengths at any time are at least 1, 1, 4, 19 and 3 processors for the benchmarks GAUSS, FFT, SOR, MSORT and TREE respectively. The entries above the highlighted ones have less than 100% utilization, and the entries below have 100% utilization.

resources can become performance bottleneck, Table 8.1 and Table 8.2 have shown that, with loop level parallelization scheduling can become the performance bottleneck even ahead of the shared bus.

Increasing the granularity of parallelism, or the amount of work for independent computing after each

Number of Processors	Bus Utilization				
	GAUSS	FFT	SOR	MSORT	TREE
2	1.9%	8.8%	6.5%	9.2%	8.3%
4	15%	14%	16%	15%	24%
8	36%	28%	41%	29%	**56%**
16	75%	**73%**	84%	72%	90%
32	**96%**	96%	**99%**	**97%**	97%
64	-	99%	99%	99%	99%

Table 8.2: Bus Utilization in the Scheduling Phase

This table gives the bus utilization in the scheduling phase. The results are meant to be compared with those in Table 8.1. In both tables entries that correspond to the configurations where the scheduling critical section starts to saturate are highlighted (in boldface).

scheduling can reduce the possibility of forming a queue (or waiting) in the scheduling critical section, but this may not always be possible. A better alternative is to find more efficient scheduling mechanisms and more efficient synchronization mechanisms (which are at the heart of the scheduling operation). This will be the main topic in Chapter 10 that evaluates the different hardware and software synchronization solutions for bus based multiprocessors.

8.3. Independent Computing Phase

8.3.1. Execution Times

The normalized execution time of the independent computing phase, and the proportion of this time to the total execution time for each benchmark are shown in Figures 8.2(a) and 8.2(b). Figure 8.2(a) indicates that perfect speedup is possible only for configurations with small numbers of processors. When enough processors are used, 32 for example, execution time is reduced only marginally when the number of processors is increased. Since the only shared resource (hardware or software) used in this phase is the bus, the performance bottleneck in this phase is the bus bandwidth, and the speedup is closely related to the bus utilization. Table 8.3 shows the speedup and the bus utilization of the independent computing phase.

In Table 8.3 the highlighted entries split all configurations into two groups: one group includes those that are above the highlighted entries, and the other group the remaining, including the highlighted entries. One prominent

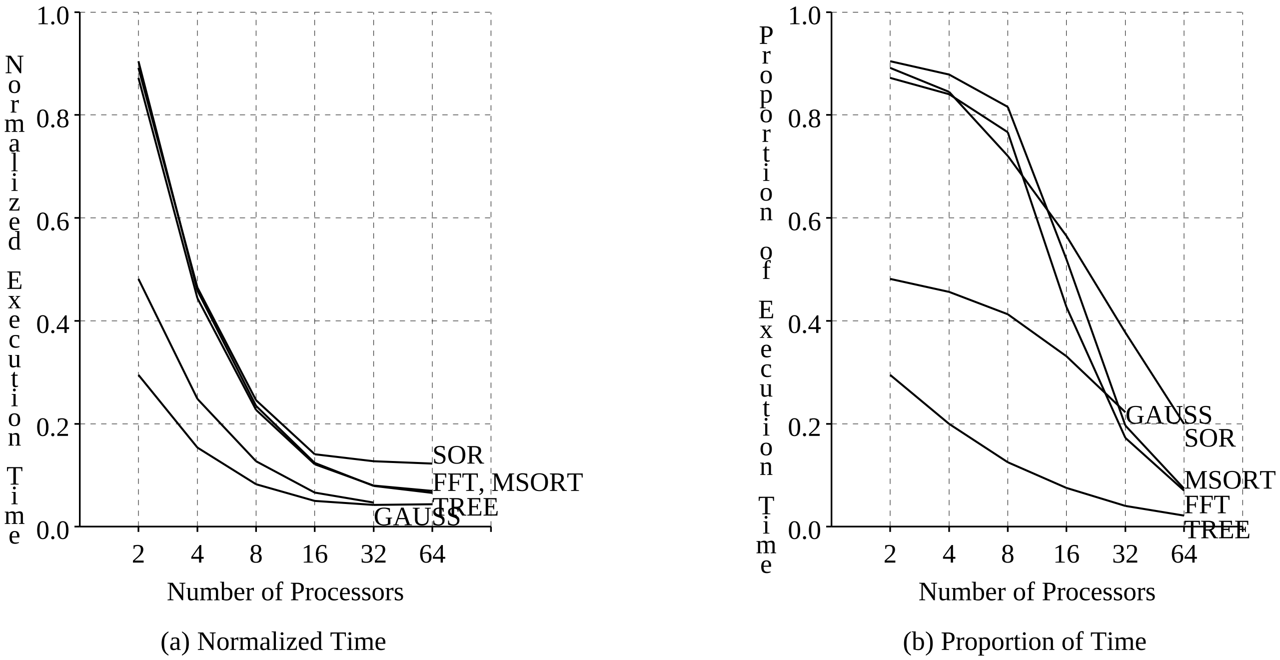


Figure 8.2: Normalized Execution Times of the Independent Computing Phase and the Proportion in Total Execution Times

Figure (a) shows the speedup characteristics of the execution time for the independent computing phase. When a smaller number of processors is used the execution time can be reduced in half if the number of processors is doubled. The speedup levels off when enough processors has already been used. However, even though the execution time of the independent computing phase barely decreases, its proportion in the total time still decrease continuously as more processors are added (Figure (b)). This is due to the increase of execution times in other computational phases, notably the scheduling phase.

difference between the two groups is that, in the first group the ratios of bus utilizations, if calculated in the same way as the ratio of execution times is calculated in this table, are all greater than 2; whereas in the second group all the ratios are smaller than 2. The fact that the ratio of bus utilizations can be greater than 2 indicates that the bus is more congested not only because the number of processors is doubled (which can make the ratio of bus utilizations at most equal to 2), but also the bus requirement of each processor has become larger. Long term bus requirement is dominated by cache miss from shared access, whose speedup characteristics will be thoroughly investigated in the next section.

8.3.2. Miss Ratio of Shared Access

Table 8.4 lists the miss ratios of shared reads and shared writes, as well as the proportion of shared reads in shared accesses. Table 8.4 indicates that the bus requirement of each processor indeed increases with the number of processors. The miss ratio of shared reads increases steadily as the number of processors is doubled. The miss ratio of shared write, on the other hand, increases toward the maximum 100% very rapidly. That nearly all shared writes are misses (write misses include invalidations, see definition of write misses in Section 6.1.1) is not surprising, because a shared data item is likely be read by multiple processors before it is modified (hence an invalidation needed). If the shared data item is not read, or read by only one processor before the data is modified, with dynamic scheduling it is very unlikely that the same part of the shared data structures is read and modified by the same processor in successive executions of a parallel loop, even with only a few processors.

Since the miss ratio of shared writes comes close to 100% when only a few processors are used, the increase in the bus requirement of each processor comes mainly from the increase in the shared read miss ratio. Although the shared read miss ratio is always smaller than that of shared writes, with a larger portion of shared accesses being reads, cache misses from shared reads can be a significant proportion of all shared misses. Table 8.5 shows the exact proportion of shared read misses in total shared misses. As indicated in Table 8.5, in almost all cases the

---

Number of of Processors	Ratio of Execution Times (RT) & Bus Utilization (%) (BU)									
	GAUSS		FFT		SOR		MSORT		TREE	
	RT	BU	RT	BU	RT	BU	RT	BU	RT	BU
2	-	3.3	-	2.6	-	3.7	-	1.8	-	8.1
4	0.52	9.6	0.51	7.8	0.52	12.4	0.51	6.0	0.52	24.2
8	0.51	24.1	0.51	19.6	0.53	33.9	0.51	17.5	0.54	56.8
16	0.52	59.2	0.54	74.8	0.57	81.3	0.53	61.9	**0.61**	**92.7**
32	**0.70**	**97.0**	**0.66**	**97.5**	**0.90**	**98.2**	**0.64**	**97.4**	0.84	98.3
64	-	-	0.87	99.4	0.97	99.7	0.83	99.2	1.03	99.2

Table 8.3: Speedup and Bus Utilization

This table shows the speedup and the bus utilization in the independent computing phase. For each configuration the speedup is represented by the ratio of the execution time of the configuration to the execution time of the configuration that has only half as many processors. For example, the execution time of the benchmark GAUSS in the independent computing phase when 32 processors are used (the entry in boldface) is 0.7 of the execution time when 16 processors are used. The highlighted (in boldface) entries indicate where the speedup has started to level off.

---

majority of shared misses come from shared reads.

The increase in the shared read miss ratio when more processors are used results from all three factors discussed earlier: dynamic scheduling, fine grain sharing and reference spreading. It would be better if the simulations can collect cache miss statistics separately for the three factors so that individual effects can be isolated, possibly

Number of of Processor	Shared Read Miss Ratio (R) & Shared Write Miss Ratio (W)									
	GAUSS		FFT		SOR		MSORT		TREE	
	R	W	R	W	R	W	R	W	R	W
2	0.05	0.51	0.07	0.89	0.14	0.84	0.07	0.95	0.10	0.91
4	0.11	0.88	0.11	0.98	0.24	0.89	0.14	0.98	0.18	0.96
8	0.16	0.94	0.14	0.99	0.37	0.90	0.21	0.99	0.31	0.97
16	0.21	0.98	0.17	1.00	0.53	0.93	0.27	1.00	0.39	0.99
32	0.27	0.99	0.20	1.00	0.62	0.93	0.30	1.00	0.45	0.99
64	-	-	0.24	1.00	0.67	0.93	0.33	1.00	0.50	1.00
Proportion of Shared Read	0.87		0.93		0.99		0.94		0.95	

Table 8.4: Miss Ratio of Shared Accesses, and the Proportion of Shared Read

This table lists the miss ratios of the two types of shared access, and the proportion of the shared reads in shared accesses (the proportion of the shared write is (1 − the proportion of shared reads)). The miss ratio of shared writes has come close to 100% with just a few processors. The miss ratio of shared reads are smaller. But since a large majority of shared access are reads, the bus requirement of shared reads, or the number of cache misses generated, is at least comparable to that of shared writes (see Table 8.5).

Note that since the total number of shared reads or shared writes in the independent computing phase remains the same no matter how many processors are used, the shared read/write miss ratio in this figure is also proportional to the number of shared read/write misses generated by each processor.

Number of Processors	Proportion of Shared Read Misses in All Shared Misses				
	GAUSS	FFT	SOR	MSORT	TREE
2	*0.40*	0.51	0.93	0.52	0.70
4	*0.45*	0.60	0.96	0.69	0.80
8	0.52	0.65	0.97	0.77	0.87
16	0.58	0.68	0.98	0.81	0.89
32	0.64	0.72	0.98	0.83	0.91
64	-	0.76	0.98	0.84	0.91

Table 8.5: Proportion of Shared Read Misses in All Shared Misses

This table shows how much likely a shared miss is contributed by a shared read. All information in this table can be calculated from Table 8.4. As indicated in all except two cases the majority of misses are originated from shared reads. Another interesting fact is that the proportion of shared read misses increases consistently as more processors are used.

making it easier to project the result for a much larger multiprocessor configuration. However, this is not possible because it is difficult to decide the cause of every cache miss in a simulation.

Although the effects of the three factors can not be distinguished, the resulting shared read miss ratio in Table 8.4 still shows a surprisingly simple and consistent trend. The trend is more obvious in Figure 8.3, which depicts the same information as in Table 8.4. In Figure 8.3(a) all lines are almost straight, indicating an approximate linear relationship between the miss ratio of shared reads and the logarithm of the number of processors. This linear relationship, however, can not be extrapolated to the cases when more than 64 processors are used, because it is apparent that the read miss ratio increases slightly less than linearly, or leveled off when 32 or more processors are used. No satisfactory explanation has been found for this simple relationship. But the relationship certainly is an

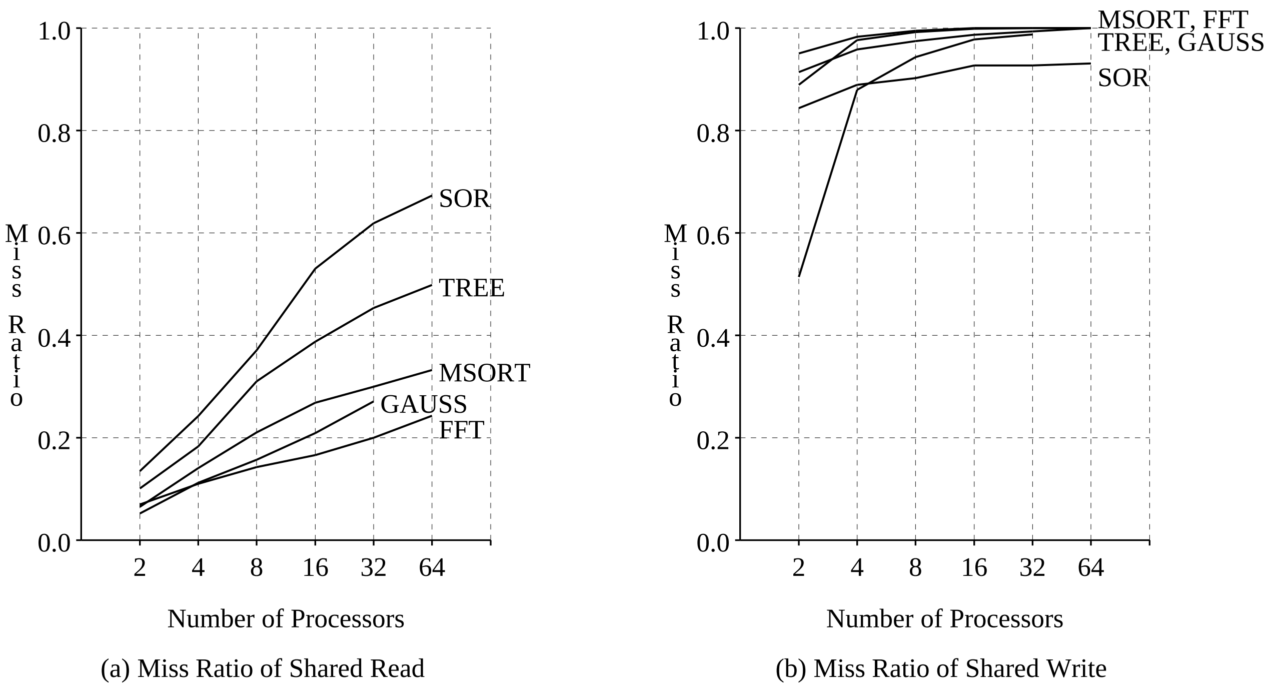


Figure 8.3: Miss Ratios of Shared Read and Shared Write

These two figures show the miss ratios of the shared reads and the shared writes in the independent computing phase, and contain the same information as Table 8.4. The miss ratio of shared reads increases steadily, in fact about linearly to the logarithm of the number of processors for some of the benchmarks.

interesting coincidence for all five benchmarks, especially because the effect of fine grain data sharing is so much problem/algorithm dependent that there is no general way to quantify it completely.

8.3.2.1. Scheduling Policy

In previous discussions, loop iterations, or units of independent computation in parallel loops are all dynamically scheduled. The alternative is to use static scheduling, which assigns a fixed number of iterations to each processor during compilation time or at the beginning of program execution. Besides the trade-off between scheduling overhead and workload balance, the scheduling policy also affects the cache miss ratio of shared access in the independent computing phase. To find the effect of using different scheduling policies, simulations are conducted for each benchmark with all the parallel loops changed from dynamic scheduling to static scheduling. The resulting miss ratios of shared reads and shared writes in the independent computing phase are shown in both Table 8.6 and Figure 8.4.

Comparing Figure 8.4(a) to Figure 8.3(a) we can see that with static scheduling the miss ratio of shared reads is significantly reduced for the benchmarks GAUSS, SOR (both by at least 45%, any configuration) and TREE (by at least 80%, any configuration). This is because in these benchmarks we are able to statically distribute the computation in the parallel loops to all processors so that each processor always, or at least most of the time, access the same set of data (including the readonly ones) during successive executions of the loops. The shared data structures in the other two benchmarks: FFT and MSORT, however, are mostly subjected to fine grain data sharing. Therefore the miss ratios of shared reads for the two benchmarks remain almost the same as if the parallel loops are dynamically scheduled.

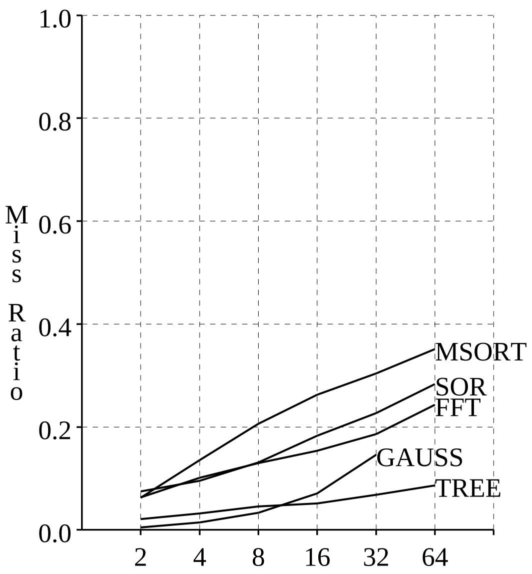
Fine grain data sharing, in fact, can be totally avoided in benchmark GAUSS with static scheduling. In this case all the shared read misses are solely attributed to reference spreading (for detail see Section 6.2.2.3), and the miss ratio of shared reads, as discussed earlier, becomes a linear function of the number of processors. The miss ratios of shared reads M_{sr} for benchmark GAUSS in Table 8.6 are accurately represented by the following equation:

$$M_{sr} = -0.00442 + 0.00471 \times P$$

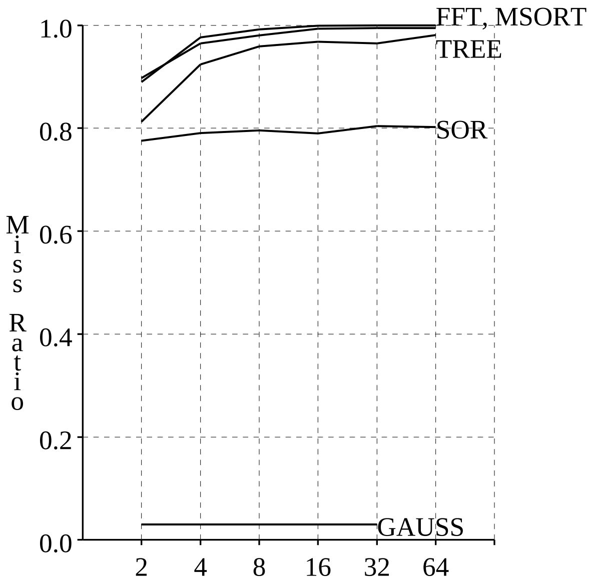
Reference spreading also exists in the other four benchmarks. But its effect is compounded by fine grain sharing and, in the case of benchmark SOR, the asynchronous nature of the accesses in the algorithm.

Number of of Processors	Shared Read Miss Ratio (R) & Shared Write Miss Ratio (W)									
	GAUSS		FFT		SOR		MSORT		TREE	
	R	W	R	W	R	W	R	W	R	W
2	0.005	0.03	0.06	0.89	0.08	0.78	0.06	0.90	0.021	0.81
4	0.014	0.03	0.10	0.98	0.10	0.79	0.14	0.96	0.032	0.92
8	0.033	0.03	0.13	0.99	0.13	0.80	0.21	0.98	0.046	0.96
16	0.071	0.03	0.15	1.00	0.18	0.79	0.26	0.99	0.051	0.97
32	0.15	0.03	0.19	1.00	0.23	0.80	0.30	0.99	0.068	0.97
64	-	-	0.24	1.00	0.28	0.80	0.35	0.99	0.087	0.98

Table 8.6: Miss Ratio of Shared Reads and Shared Writes



(a) Miss Ratio of Shared Read



(b) Miss Ratio of Shared Write

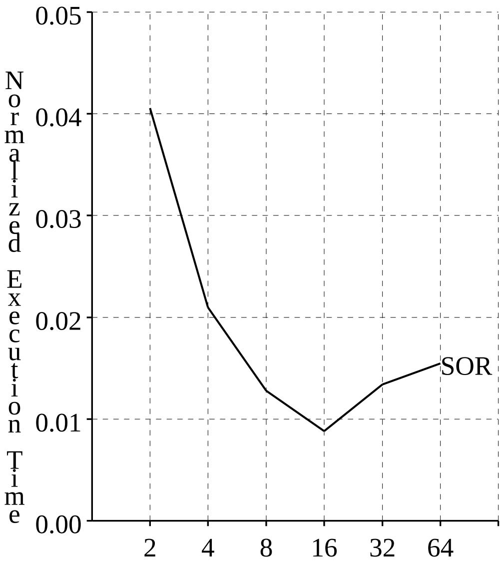
Figure 8.4: Miss Ratio of Shared Reads and Shared Writes

Both Table 8.6 and Figure 8.4 show the miss ratio of shared reads and shared writes in the independent computing phase. The results are derived from the same parallel loops as in Table 8.4 and Figure 8.3, except here the parallel loops are all *statically* scheduled.

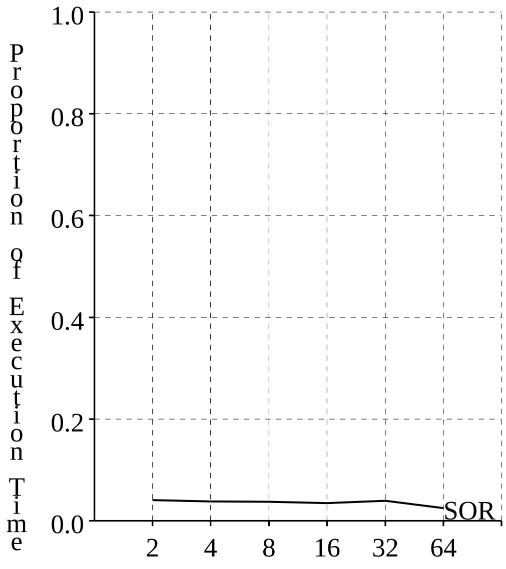
The switch from dynamic to static scheduling also reduces the miss ratio of shared writes, as evident from comparing Figure 8.4(b) to Figure 8.3(b). But the amount of reduction is quite different for different benchmarks. The most dramatic change occurs on benchmark GAUSS, where the the miss ratio of shared writes not only is significantly reduced, bus also has become independent of the number of processors. The reductions in miss ratios in other benchmarks are rather small, indicating that most shared data structures that are frequently modified in these benchmarks are still subjected to fine grain sharing.

8.4. Communication Phase

The normalized execution time of the communication phase, and the proportion of the time in the total execution time for the benchmark SOR are shown in Figures 8.5(a) and 8.5(b). Only the line for SOR is shown because it is the only benchmark that has a communication phase. The speedup characteristics of the execution time for communication phase is quite similar to that for the scheduling phase (compare Figure 8.5 to Figure 8.1). This is not



(a) Normalized Time



(b) Proportion of Time

Figure 8.5: Normalized Execution Times of the Communication Phase and the Proportion in Total Execution Times

The figures show the speedup characteristics of execution time in the communication phase. Since the communication critical section is rather small in the benchmark SOR, which is the only program that has a communication phase (see the program code of SOR in Figure 7.1), the proportion of execution time is rather insignificant.

surprising since communication as well as scheduling is implemented with a small critical section. In both cases the number of times a processor goes through the critical section is inversely proportional to the number of processors. Thus if bus utilization is low and no waiting occurs at the critical section, the execution time of the communication phase is halved when the number of processors is doubled. As more processors are added the execution time starts to increase, because the cache miss latency becomes longer and the processors are more likely to accumulate in the critical section. The latter certainly is not a serious problem here, because the communication critical section in SOR is quite small and a long queue is unlikely to form on it. So the proportion of time the processor spends in this phase is always small.

8.5. Problem Size

In this section the effect of problem size on the speedup characteristics of some performance measures are discussed. The main objective is to find the trend of the speedup characteristics when the problems size is increased, and from the trend to predict the performance of the benchmarks when the problem size is much larger. The two important aspects of parallel computation that can be directly affected by the change in problem size are the *granularity of parallelism*, and the *cache miss ratio* in the independent computing phase, as explained in the following two sections.

8.5.1.1. Granularity of Parallelism

The granularity of parallelism is the size of a unit work that is assigned to a processor each time it goes through the scheduling critical section. To minimize the scheduling overhead we like to have the largest possible scheduling granularity. However, to achieve a better workload balance among all processors, and consequently better speedup, we also like to have smaller granularity. With loop level parallelization and the use of a simple dynamic self-scheduling method, each iteration in a parallel loop is a natural choice of scheduling unit. In this section the effect of changing the problem size on the amount of computation in an iteration of a parallel loop will be analyzed.

Whether the amount of computation in an iteration of a parallel loop will increase with the problem size can be deduced by examining the original parallel code. For the benchmarks GAUSS and SOR the amount of computation in a loop iteration is found to grow with $O(N)$ (because the innermost loop has $O(N)$ iterations and it is the *second* innermost loop that is parallelized), and for MSORT, $O(\log N)$ (because in each iteration binary search is conducted on an array of size N). For the other two benchmarks the problem size has no effect.

With a larger scheduling unit the most important consequence is that the number of processors that can simultaneously be in the independent computing phase becomes larger. This information is shown in Table 8.7, for both the small and the larger problem sizes.

As expected, from Table 8.7 we can see that more processors can engage in independent computing at the same time for the benchmark GAUSS, SOR and MSORT when the problem size becomes larger. The trend can continue indefinitely for these benchmarks, i.e., with a large enough problem size all the processors can be made busy doing independent computing. This does not imply that the speedup can increase indefinitely, because eventually it will be limited by the bus bandwidth. However, as the next section shows, sometimes the problem size can affect the bus requirement of a processor in the independent computing phase, and thus the maximum speedup of a parallel program.

For benchmarks FFT and TREE the number of processors that can simultaneously be in the independent computing phase changes little with problem size. A small decrease in the number of processors, however, is observed in benchmarks FFT and TREE when 32 or 64 processors are used. But for all except the 64-processor FFT case, the decrease is so small that it is within the range of the random fluctuation of the simulation result. The more significant decrease observed in the 64-processor TREE benchmark is attributed to the reduction of the cache miss

Number of	Smaller & Larger Problem Size									
Processors	GAUSS		FFT		SOR		MSORT		TREE	
2	1.9	1.9	1.8	1.8	1.8	2.0	1.8	1.9	1.1	1.1
4	3.8	3.9	3.6	3.6	3.7	4.0	3.6	3.7	1.2	1.2
8	7.2	7.7	7.0	7.1	7.2	7.9	6.9	7.3	1.3	1.3
16	12	15	8.1	8.3	13	16	9.2	12	1.5	1.5
32	14	22	6.6	6.5	16	31	7.0	9.6	1.5	1.4
64	-	-	5.3	4.6	16	51	5.3	6.0	1.6	1.4

Table 8.7: The Number of Processors that Can Simultaneously in the Independent Computing Phases

This table lists the average number of processors that can simultaneously be in the independent computing phase, for the small and the larger problem sizes. For benchmark GAUSS, SOR and MSORT the granularity of parallelism, or the size of the scheduling unit increases when the problem size becomes larger. Therefore the number of processors that can simultaneously engage in independent computing increases when the problem size becomes larger.

ratio in the independent computing phase (detail is in the next section). When the bus access latency is significant (e.g., as in the case of 64 processors), with lower miss ratios processors are more likely to finish the independent computing phase and return to the scheduling critical section earlier. Therefore the number of processors that can simultaneously be in the independent computing phase can become smaller.

8.5.1.2. Cache Miss Ratio

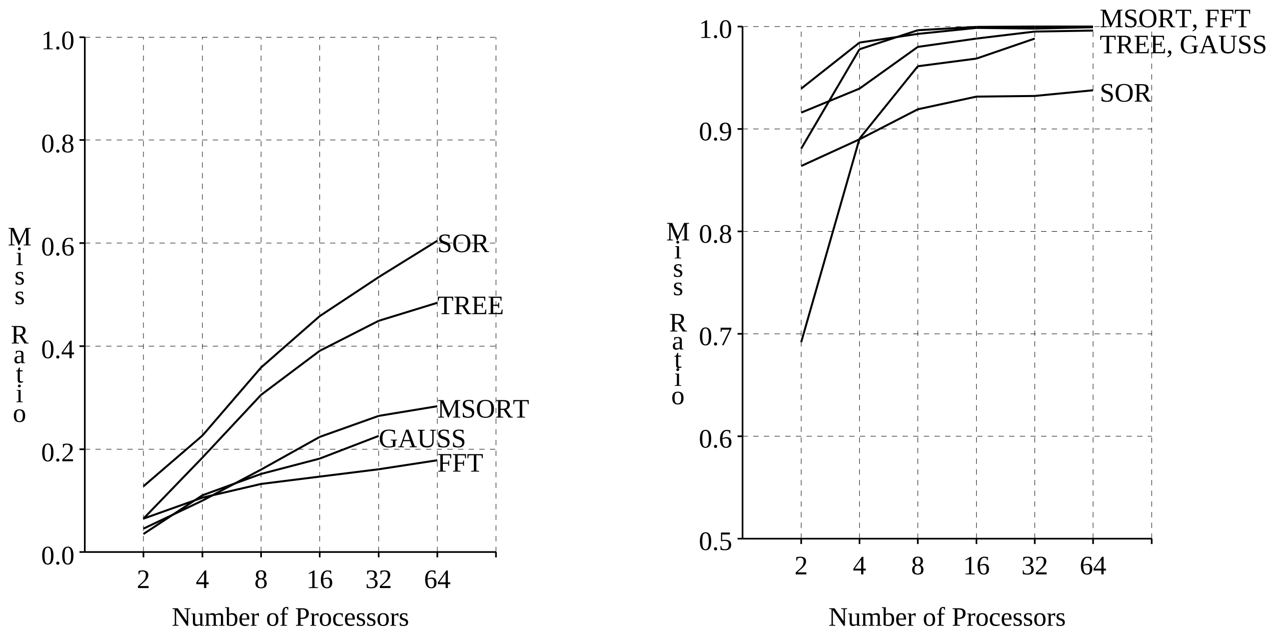
The problem size can affect the cache miss ratio in the independent computing phase. When the problem size is larger, cache misses from non-shared data access becomes less significant, because usually the total number of cache misses from non-shared data access does not change with the problem size. A larger problem size can also reduce the miss ratio of shared accesses. For example, if increasing the problem size can increase the number of iterations in a parallel loop, the miss ratio of accessing the loop invariants will be smaller. This is because each processor can now work on more iterations, and only in the first iteration does reading the loop invariants causes cache misses.

The miss ratios of shared reads and shared writes in the independent computing phase for the larger problem sizes are shown in Figure 8.6. By comparing Figure 8.6(a) to Figure 8.3(a) (also see Table 8.8, for easier comparison) we can see that the shared read miss ratios of all but the benchmark TREE decrease as the problem size becomes larger. The shared read miss ratio of the benchmark TREE does not change because no loop invariant exists in the parallel loop of the benchmark.

The effect of changing the problem size on the miss ratio of shared writes is relatively small, especially when more processors are used (comparing Figure 8.6(b) to Figure 8.3(b)), because the change in problems size does not affect the fine grain sharing nature of the data structures that are modified frequently in the programs.

The effect of increasing the problem size on cache miss ratio also depends on the scheduling policy. If *static scheduling* is used and there is *little fine grain sharing*, cache miss ratio (shared read or shared write) can decrease if the *amount of computation grows faster than the data size*. This is because the number of misses increases with the data size, and the number of accesses increases with the amount of computation. With little data migration the miss ratio will be proportional to the ratio of the amount of computation to the data size.

Increasing the problem size can increase the ratio of the amount of computation to the data size for benchmarks GAUSS, FFT and MSORT. But since only the benchmark GAUSS has little fine grain sharing when static scheduling is used, only the benchmark GAUSS has a smaller shared read miss ratio when the problem size



(a) Miss Ratio of Shared Reads

(b) Miss Ratio of Shared Writes

Figure 8.6: Miss Ratio of Shared Reads and Shared Writes

These two figures show the miss ratios of shared reads and shared writes during the independent computing phase when slightly larger problem sizes are used. As in the case of Figure 8.3 all parallel loops are dynamically scheduled.

Number of Processors	Ratio of Sh. Read Miss Ratios (RR) & Ratio of Sh. Write Miss Ratios (RW)									
	GAUSS		FFT		SOR		MSORT		TREE	
	RR	RW	RR	RW	RR	RW	RR	RW	RR	RW
2	0.67	1.3	0.93	1	0.94	1	0.70	1	0.65	1.0
4	0.98	1.0	0.96	1	0.93	1	0.71	1	1.0	0.98
8	0.96	1.0	0.93	1	0.97	1	0.76	1	0.98	1.0
16	0.87	0.99	0.88	1	0.86	1	0.83	1	1.0	1.0
32	0.83	1.0	0.80	1	0.86	1	0.88	1	0.99	1.0
64	-	-	0.73	1	0.90	1	0.85	1	0.97	1.00

Table 8.8: Ratio of Shared Read Miss Ratios and Ratio of Shared write Miss ratios of the Two Problem Sizes

This table facilitates the comparisons between Figure 8.6, which shows the miss ratio of shared accesses in independent computing phase with the problem size larger than in Figure 8.3, to Figure 8.3. Each number in this table is the ratio between the corresponding miss ratio in Figure 8.6 and Figure 8.3. For example, when 32 processors are used with the benchmark GAUSS, the shared read miss ratio is 0.23 in Figure 8.6, and 0.27 in Figure 8.3. The ratio of these two miss ratios is 0.23 / 0.27 = 0.83, as shown in this table.

becomes larger. The total amount of computation in the independent computing phase for program GAUSS is $O(N^3)$, and the data size $O(N^2)$. So the miss ratio of shared access is proportional to $O(\frac{1}{N})$. Table 8.9 confirms the prediction. One interesting result of this decrease in miss ratio for the benchmark GAUSS is that, since the bus requirement of each processor is inversely proportional to the problem size, the maximum speedup of the benchmark becomes proportional to the problem size.

8.6. Summary and Conclusions

In this chapter the speedup characteristics of some performance measures in scheduling, independent computing and communication phases are analyzed. The purpose of the study is two folds: first, to determine that, if loop level parallelization is adopted, whether scheduling can become a performance bottleneck; second, if the bus becomes the performance bottleneck, to find out what the speedup characteristics of the bus requirement of each processor are, and how the bus requirement is affected by other factors such as scheduling policy and problem size.

It should be noted that all the conclusions stated here are derived from very limited number of multiprocessor configurations with 5 sets of parallel benchmarks. The only parameter in the multiprocessor configuration that has been changed is the number of processors. Others such as the speeds of processors, caches, the shared bus and main memory, as well as the cache block size and the bus switching method are all fixed. The simulation results may be strongly affected by the choice of certain parameter values, e.g., the cache miss ratios by the choice of the 4 byte

Number of Processors	Cache Miss Ratio & Ratio of Cache Miss Ratios			
	Shared Read		Shared Write	
2	0.52%	0.52	2.30%	0.51
4	0.85%	0.52	2.66%	0.50
8	1.53%	0.53	2.84%	0.50
16	2.87%	0.53	2.94%	0.50
32	5.55%	0.53	2.99%	0.50

Table 8.9: Cache Miss Ratio and Ratio of Cache Miss Ratios of the Two Problem Sizes

This table shows the cache miss ratio of shared reads and shared writes in the independent computing phase for the benchmark GAUSS, when the problem size is 64 (i.e., the matrix is 64 by 64) and static scheduling is used. The table also shows the ratio of the miss ratio of the larger problem size to the miss ratio of the smaller problem size, when both uses the static scheduling methods. As predicted the cache miss ratios drop in half when the problem size is doubled.

cache block size. Therefore the following conclusions are valid for the multiprocessor configurations that have been simulated and the parallel benchmarks used, and may be true only to a limited extent for other multiprocessor configurations and parallel programs. Interested reader should read Section Section 7.1 to understand the basic structure of the multiprocessor simulated, Section 7.3 for the parameter values chosen for the multiprocessor, and Section 7.2 of the 5 parallel benchmarks used in the simulation.

In the first study scheduling is found to become the performance bottleneck for the benchmarks FFT, MSORT and TREE even though the bus is not saturated. In the second study the total bus requirement from all processors during the independent computing phase is found to increase with the number of processors. The increase in bus requirement is attributed to three factors: fine grain data sharing, reference spreading, and dynamic scheduling policy.

Shared data structures that are subjected to fine grain data sharing have high cache miss ratios, no matter what scheduling policy is used. For data structures that are not subjected to fine grain sharing the use of static scheduling can reduce the cache miss ratio. The effect can be dramatic, as in the case of benchmark GAUSS, where most of the time the major data structures are not actively shared by multiple processors if static scheduling is used. Reference spreading is related to a special kind of shared variables, i.e., the loop invariants in a parallel loop. Since every processor has to load one copy of loop invariants, the miss ratio of loading loop invariants is proportional to the number of processors.

It is difficult to quantify completely the effect of fine grain data sharing, since the sharing pattern can be complex and very much problem/algorithm dependent. Therefore it is even more difficult to explain the combined effect of all these three factors on the increase in bus demand. Yet the simulation results show a simple relationship that the shared read miss ratio grows about linearly to the logarithm of the number of processors. This simple relationship is still preserved when the problem size becomes larger.

Finally, changing the problem size can affect the granularity of parallelism and the cache miss ratio. If the granularity of parallelism is increased with the problem size, as in the case of benchmarks GAUSS, SOR and MSORT, scheduling is less likely to become the performance bottleneck. Increasing the problem size can also reduce the cache miss ratio. For example, the miss ratio of reading loop invariants is inversely proportional to the problem size. If fine grain sharing can be totally avoided by the use of static scheduling, and the amount of computation grows faster than the data size when the problem size increases, as in the case of benchmark GAUSS, the maximum speedup of the program can increases with the problem size.

It is apparent that the performance bottleneck of executing a parallel loop can either be the *bus bandwidth* or *scheduling*. The bus bandwidth problem can be dealt with by increasing the bus bandwidth, or reducing the bus demand from the caches. Bus bandwidth can be increased with more investment in hardware such as using a wider and faster bus. Bus demand can be reduced by software means such as using static instead of dynamic scheduling, though this runs the risk of creating an unbalanced workload for processors.

Dynamic scheduling is always important not only because it guarantees a more balanced workload under any input condition, but also because it is more adaptable to the unequal progress of processors created by the scheduler of the operating system. However, the overhead of scheduling can be so great that sometimes it becomes the performance bottleneck. Since a scheduling critical section implemented with test&test&set can generate a large amount of unnecessary bus traffic, there is plenty of room for improvement on scheduling. This will be the subject of Chapter 10.

Chapter 9
Barrier, Barrier-Spin Wait and Serial Phases

9.1. Introduction

In this chapter the speedup characteristics of the barrier, barrier-spin wait and the serial phases are studied. The barrier and the barrer-spin wait phases are related to the barrier operation, which, besides scheduling, is an important parallel processing overhead. Computation in the serial phase, on the other hand, is not a parallel processing overhead. However, it still limits the maximum speedup of a parallel program, as prescribed by Amdahl's law.

The barrier operation increases execution time in two ways. First, every processor has to perform an atomic decrement (or increment) on a barrier counter. Second, processors have to wait until all processors have reached the barrier. In my simulation model an *atomic decrement* instruction is assumed available, as in Symmetry multiprocessors. The simulation results will show that, a very efficient atomic instruction makes the first cost of the barrier operation nearly negligible. The alternative to the atomic instruction is a much more cumbersome software means of critical section. The simulation will show that if implemented with a critical section, the execution time of a barrier operation can be substantially larger than when the atomic increment instruction is used.

The execution time of the barrier-spin wait phase is mostly affected by the difference in the arrival times of processors at the barrier, or equivalently the difference in the finishing times of processors in the independent computing phase. Balancing the workload can reduce the barrier-spin wait time, but achieving the balance, for example, by utilizing a smaller granularity of parallelism, will increase the cost of scheduling. Also, the use of dynamic scheduling itself can become a major cause of the different arrival times at the barrier. The barrier-spin wait time can increase rapidly with the number of processors when a queue has already formed in the scheduling critical section.

The execution time of the serial phase can also increase with the number of processors, even though the total amount of serial computation is fixed. This is because, as will be shown in the simulation results (Section 9.5), the total number of shared misses in the serial phase is close to a linear function of processors.

The barrier operation is detrimental to the speedup of parallel execution, and the computation in the serial phase is impervious to speedup with multiprocessing. Fortunately, the speedup prospect is not as bleak as, for example, painted by Amdahl's law. The number of times the barrier operation is performed and the amount of computation in the serial phase both grow slower than the total amount of computation in a parallel program, as the problem size increases. When the problem size is large enough to warrant the use of a multiprocessor, the execution times of these three phases tend to become insignificant.

9.2. Barrier Phase

The normalized execution time of the barrier phase, and the proportion of the time in the total execution time is shown in Figure 9.1. The only operation in the barrier phase is an atomic increment instruction, which makes one

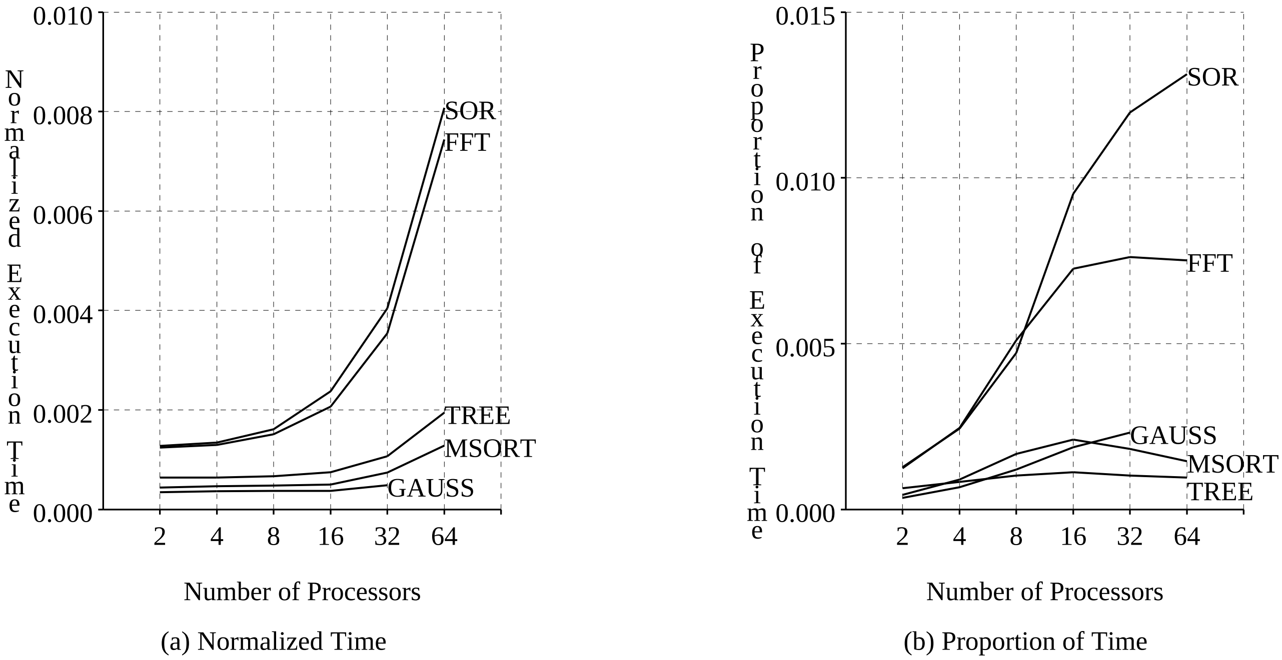


Figure 9.1: Normalized Execution Times of the Barrier Phase and the Proportion in The Execution Times

Figure 9.1(a) shows the normalized execution time of the barrier phase. When the number of processors are only a few, e.g. 2 to 16, the execution time changes very little as the number of processors increases. But when more processors are used the execution time increases rapidly due to the longer bus access time. See Table 9.1 for the bus utilization in this phase. Figure 9.1(b) shows the proportion of the barrier time in the total execution time. The proportion of the time becomes larger initially when more processors are added to the two processor system, because the total execution time is decreasing at the same time. But when many processors are used the proportion of barrier time starts to level off, or even decreases as the number of processors increases, because the scheduling time has increased in a even faster rate, (see Figure 8.1).

Number of	Bus Utilization				
Processors	GAUSS	FFT	SOR	MSORT	TREE
2	4.4%	10%	10%	5.2%	8.8%
4	7.0%	19%	20%	7.7%	17%
8	15%	37%	41%	16%	31%
16	30%	69%	72%	44%	55%
32	74%	91%	92%	80%	79%
64	-	96%	98%	93%	93%

Table 9.1: Bus Utilization in Barrier Phase

This table is used to explain the increase in the barrier time of Figure 9.1. The increase is related to the bus access time, or bus utilization. When the bus utilization is high, e.g., greater than 80%, increasing the number of processors can increase the execution time of the barrier phase rapidly.

bus access. Since the atomic operation is short, processors are not likely to queue up for the operation. So the execution time of this phase increases with bus utilization (see Table 9.1). However, even at its largest the proportion of the barrier time in the total execution time is still very small (smaller than 1.5%, as shown in Figure 9.1(b)).

9.3. Barrier-Spin Wait Phase

The normalized execution time of the barrier-spin wait phase, and the proportion of the time in the total execution time is shown in Figure 9.2. The increase in the barrier-spin wait time for the benchmark FFT, MSORT and TREE when the number of processors increases is closely related to the scheduling rate in the scheduling phase. All these benchmarks utilize smaller granularities of parallelism than GAUSS or SOR. So for these benchmarks scheduling queues form when smaller numbers of processors are used (see Table 8.1). As soon as a scheduling queue forms the difference in the arrival times at the barrier will be largely determined by the time for the scheduling critical section to schedule all processors (see Section 6.1.7). The scheduling time can increase rapidly as the scheduling queue becomes longer, so the barrier-spin wait time also increases rapidly with the number of processors.

The increase in the barrier-spin wait time for the benchmarks GAUSS and SOR is less related to scheduling because a larger granularity of parallelism is utilized in the programs. In both cases a scheduling queue does not form until about 32 processors are used (see Table 8.1). When there are fewer than 32 processors the difference in the arrival times at a barrier is more related to the *variance* in the size of the scheduling unit, or the amount of

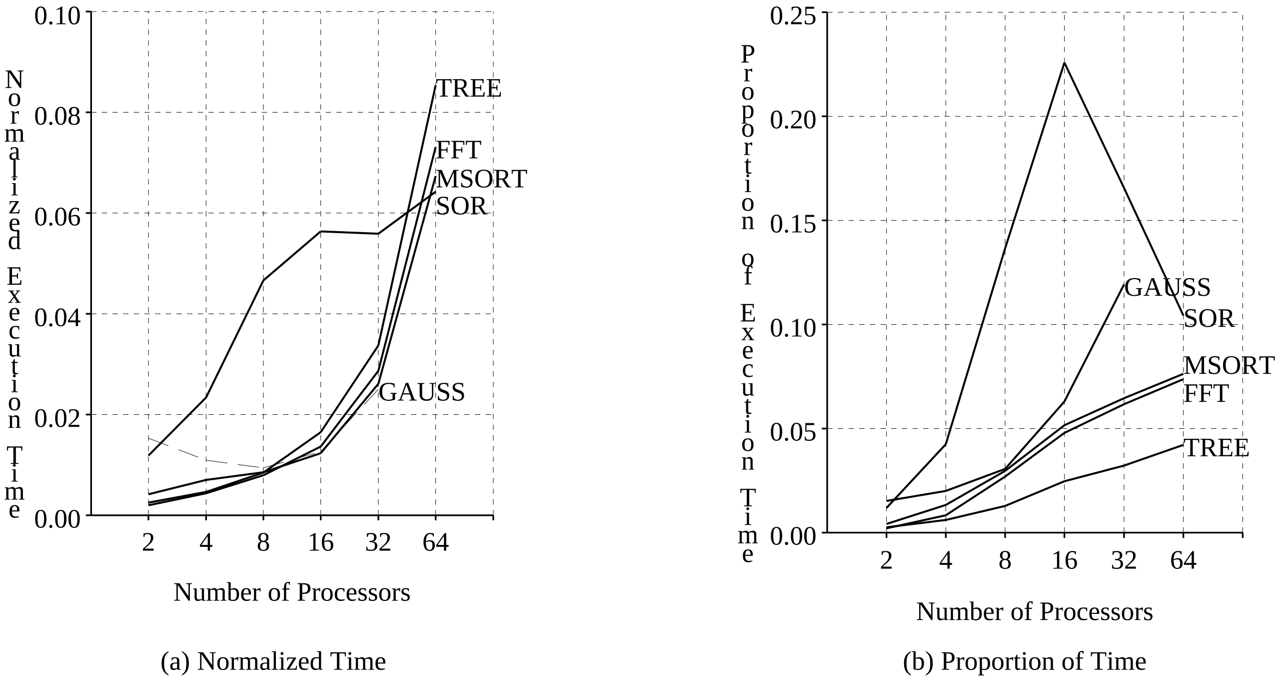


Figure 9.2: Normalized Execution Times of the Barrier-Spin Wait Phase and the Proportion in Total Execution Times

Figure 9.2(a) shows the normalized execution time of the barrier-spin wait phase. There are two groups of lines in 9.2(a): the first group includes MSORT, FFT and TREE, and the second GAUSS and SOR. In the first group the execution times always increase as processors are added, and increase more rapidly as more processors are used. In the second group the barrier-spin wait time of GAUSS decreases initially and then starts to increase, while the time of SOR has a more irregular pattern of increase when the number of processors increases. These different speedup behaviors are related to the different *granularity of parallelism* utilized in the programs. The detail can be found in the section text and in Chapter 6. Figure 9.2(b) shows the proportion of barrier-spin wait time in the total execution time. Although most of the time the proportion is less than 10%, it can be as high as 22%.

computation in a loop iteration. For the benchmark GAUSS this variance is small, and processors are likely to take an equal number of iterations even though dynamic scheduling is used. So the situation is similar to when the loop is statically scheduled, where the barrier-spin wait time decreases as more processors are used (detail is also in Section 6.1.7). For the benchmark SOR the variance in the amount of computation within a loop iteration is very large. In fact some iterations may not even contain any computation at all (program SOR uses a sparse matrix). So different processors can take unequal numbers of iterations in different executions of the parallel loops. Thus the difference in the finishing times of independent computing, the average barrier-spin wait time, can be very irregular when the number of processors is increased.

The proportion of barrier-spin wait times in the total execution times is shown in Figure 9.2(b). As indicated by the figure the proportion of barrier-spin wait time can be as substantial as 22% of the total execution time. The figure also depicts the trade-off between granularity of parallelism (scheduling time) and balance of work (barrier-spin wait time). The two benchmarks GAUSS and SOR that utilize larger granularities of parallelism spend a larger proportion of time in the barrer-spin wait phase.

9.4. Using a Critical Section To Implement the Barrier Operation

If the atomic decrement instruction is not available the barrier operation may need to use a more time consuming software critical section. Figure 9.3 shows the proportion of barrier time (only the time to decrement the

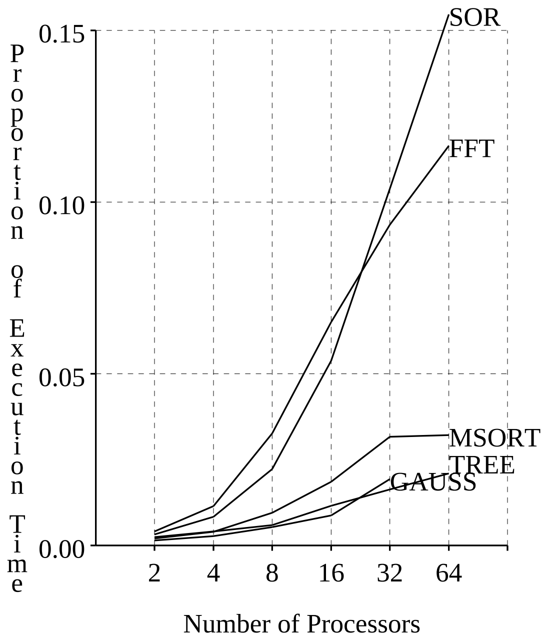


Figure 9.3: The proportion of Barrier Time in Total Execution Time

This figure shows the proportion of the execution time of the barrier phase in the total execution time. The figure is similar to Figure 9.1(b), except that in this figure programs use a *critical section* instead of an atomic decrement instruction to implement barrier operations. Note the different scales used in Y-coordinates in Figure 9.1(b) and this figure. The proportion of the barrer time increases by about 10-fold from the use of a less efficient critical section.

barrier counter), when a critical section instead of an atomic decrement instruction is used to implement the barrier. The proportion of the execution time is found to increase by about 10-fold (compare Figure 9.3 to Figure 9.1(b)). Therefore it is very important for a shared bus multiprocessor to have some form of atomic increment/decrement instructions.

9.5. Serial Phase

The normalized execution time of the serial phase, and the proportion of the time in the total execution time is shown in Figure 9.4. Although the amount of computation in the serial phase does not change with the number of

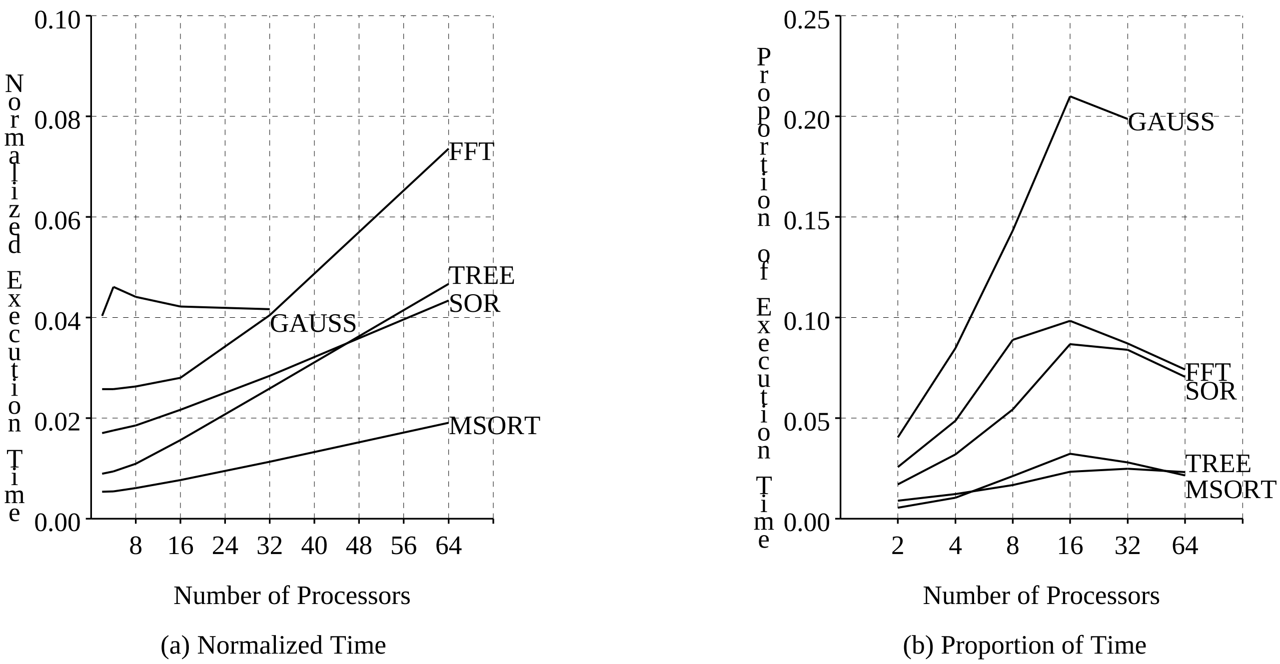


Figure 9.4: Normalized Execution Times of Serial Phase and the Proportion in Total Execution Times

Figure 9.4(a) shows the normalized execution time of the serial phase. Even though there is a fixed amount of computation in the serial phase, the execution times of all except benchmark GAUSS approximate a linear function of the number of processors when many processors are used. Figure 9.4(b) shows the proportion of the time of the serial phase in the total execution time. The proportion increases initially when more processors are used, as prescribed by Amdahl's law. The proportion eventually falls off because the scheduling time has been increasing even faster.

processors, the execution time of the serial phase still can increase when more processors are used. This is because the number of shared misses in the serial phase increases with the number of processors. In fact the total number of the shared misses is a linear function of the number of processors (see Figure 9.5).

Since the portion of the shared misses that are proportional to the number of processors (i.e., excluding the ‘‘constant’’ portion of accesses, which are issued by the processor that does the serial computing) tends to be issued by *all* processors at the same time, their cache miss latencies are almost proportional to the number of processors (see Section 6.2.6). So when the bus access time becomes a dominant part of computation in the serial phase, as in the case of all except the benchmark GAUSS when 16 or more processors are used, the execution time also approximates a linear function of the number of processors.

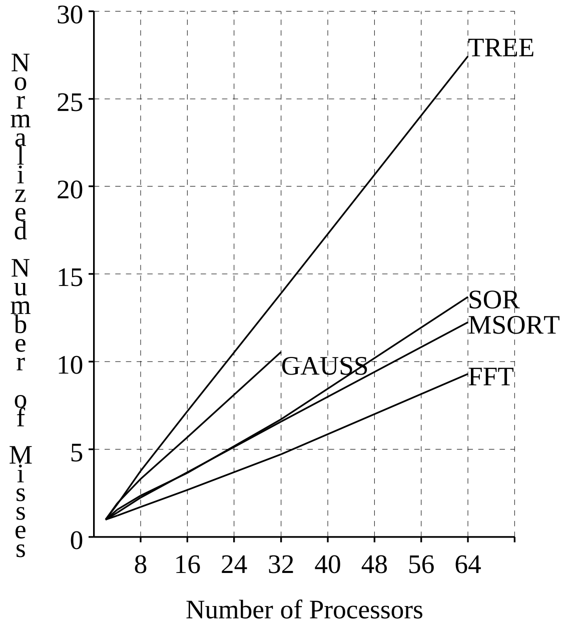


Figure 9.5: Normalized Total Number of Shared Misses in the Serial Phase

This figure shows the normalized number of shared misses in the serial phase. The normalizing base for each benchmark is the number of the shared misses when two processors are used. Figure 9.5 indicates that the number of shared misses in the serial phase is close to a linear function of the number of processors.

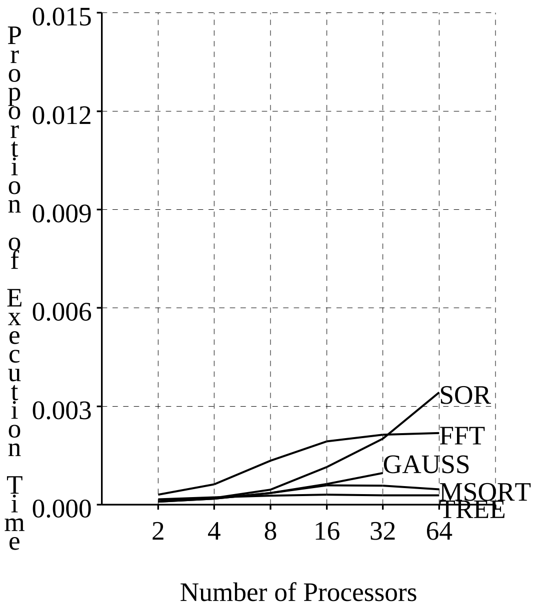
The execution time of GAUSS in the serial phase is relatively constant, because the program has relatively longer ‘‘one-processor’’ computation during the serial phase than others (as indicated by 9.4(b)), and is less affected by the increase in the memory access time. However, a slight decrease in the execution time is observed when the number of processors is increased from 2 to 4, and 4 to 8. This is because the *barrier-spin wait time* component of the serial time is more significant in GAUSS. The serial phase in GAUSS comes right after a parallel loop that is *statically* scheduled, while the serial phases in all other benchmarks come after, most of the time, a barrier operation. Since a serial phase itself is always started with a barrier operation (called the S-barrier operation, to be distinguished from barrier operations in any other place), the barrier-spin wait time component (due to the S-barrier operation) in the serial phase is always small because the earlier barrier operation (not the S-barrier operation) has *synchronized* all the processors. But in the case of GAUSS the S-barrier operation in the serial phase immediately comes after a parallel loop. So its barrier-spin wait time can be significant. Since the preceding parallel loop is statically scheduled, the barrier-spin wait time can decrease with the number of processors (see Section 6.2.5).

9.6. Larger Problem Size

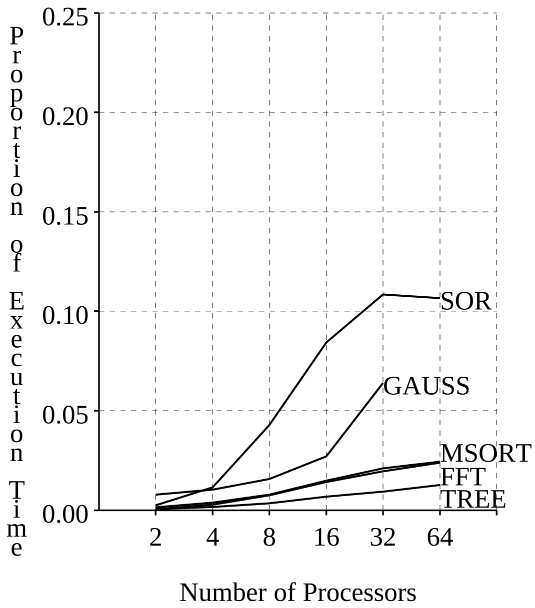
Since the number of times the barrier operation is performed and the amount of computation in the serial phase both grow slower than the total amount of computation in a parallel program, when the problem size increases the proportion of the execution time of the three phases should decrease if a larger problem size is used. This prediction is confirmed in Figure 9.6. Figures 9.6(a), (b), (c) show respectively the proportions of the execution times of the barrier, the barrier-spin wait and the serial phases in the total execution time. These figures are similar to Figure 9.1(b), Figure 9.2(b) and Figure 9.4(b), only that the results here are derived from a *larger problem size*. Each subfigure in Figure 9.6 is meant to be compared with the corresponding one in the three earlier figures. To make the comparison easier the scale in the Y-coordinate is the same for each related pair of figures. It is apparent from the comparison that the proportions of execution times of all three phases decrease as the problem size becomes larger.

9.7. Summary and Conclusion

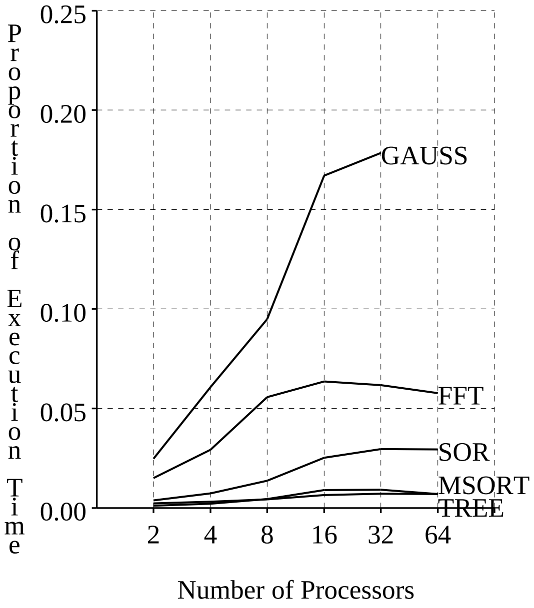
In this chapter the speedup characteristics of the barrier, barrier-spin wait and the serial phase are discussed. The execution times of all three computational phases can increase with the number of processors, especially rapidly in the case of the barrier and barrier spin wait phases. But fortunately the proportions of the execution times in the total execution time decrease as the problems size increases.



(a) Barrier Phase



(b) Barrier-Spin Wait Phase



(c) Serial Phase

Figure 9.6: The Proportion of Execution Times in the Total Execution Time

In the barrier phase a processor decrements a barrier counter atomically. The execution time of the barrier phase, though increasing with the number of processors, is very small when an atomic decrement instruction is available. But if the less efficient software implementation of critical sections is used, the execution time can increase substantially. The simulation result shows that the increase is about 10-fold.

The execution time in the barrier-spin phase is determined by the granularity of parallelism (or the scheduling unit), the variance in the size of the scheduling unit, and the scheduling strategy used in the programs. If the granularity, or the amount of computation in a loop iteration is small, the barrier-spin wait time is more related to the time for the scheduling critical section to schedule all processors. If the granularity of parallelism is large, the speedup characteristics of the execution time can be irregular, depending on the variance in the size of loop iterations.

Lastly, the execution time of the serial phase also increases with the number of processors (with one or all processors carrying out the serial computation), even though only a fixed amount of computation is carried out in this phase. The execution time is a linear function of the number of processors when the bus access time becomes the dominant part of the execution time.

Chapter 10
Synchronization

10.1. Introduction

Synchronization in parallel processing coordinates the use of shared resources. In the five benchmarks studied synchronization takes place in barrier and critical section operations (e.g., to implement dynamic scheduling). Simulation results from Chapter 8 have shown that the overhead of dynamic scheduling, when the scheduling critical section is implemented with test&test&set (TTS), can be so large that it severely reduces the potential speedup of a parallel program. In some cases, contention for the scheduling critical section became the bottleneck even before the bandwidth of the shard bus.

The inefficiency of scheduling is a result mainly from the shortcomings of the TTS implementation. First, the scheduling rate is low because quite a few instructions are executed and memory accesses made to go through the critical section. Second, when the scheduling rate is very low, and processors have to wait in the scheduling queue, the waiting processors can generate a large amount of bus traffic (see Section 6.2.1). This bus traffic reduces the bus bandwidth available to processors doing useful independent computation, thereby increasing the corresponding computation time.

Several approaches can be taken to improve the performance of scheduling. First, the basic TTS can be improved by hardware enhancements to reduce bus traffic. Second, the two primitives *test&set* and *test* from which TTS is composed can be used to implement a more sophisticated critical section that reduce the number of times these primitives are executed, and hence the bus requests generated. Third, a completely different synchronization method can be developed to replace TTS, or even the entire critical section. In this chapter the performance of the *Software Queue* [6], the *Lock Table*, and the *Restricted Fetch&Add* [63] methods will be evaluated. The software queue method belongs to the second approach, while the other two methods belong to the third.

The rest of this chapter is organized as follows. Section 10.2 discusses several examples that correspond to the three approaches mentioned above to improve the performance of synchronization. In particular the function of a software queue, the lock table and the restricted fetch&add (with combining) methods are described. Since the lock table is a new synchronization method proposed in this thesis, its implementation issues are also mentioned. The performance of the software queue, the lock table, and the restricted combining methods are evaluated in Section 9.3. Finally a summary of the evaluation is given in Section 9.4.

10.2. Alternative Synchronization Methods

10.2.1. Hardware Enhancement of TTS

A large number of bus accesses are generated as a processor enters or leaves a critical section with a long waiting queue, if the critical section is implemented with TTS. Various hardware measures were proposed to minimize the number of these accesses. For example, *write* or *read broadcast* reduces the number of reads [37, 57], and *test&set abandonment* reduces the number of test&set bus requests [28]. With a combination of these measures the number of bus accesses generated by scheduling one processor can be reduced from $O(p)$ to $O(1)$, where p is the queue length of, or the number of processors waiting to access, the critical section. However, unnecessary write or read broadcasts can be triggered if the system can not distinguish between the accesses to the lock and the non-lock variables. Thus a system with write broadcast may as a whole generate more bus traffic, and the use of caches by processors can be unnecessarily disrupted by both write and read broadcasts.

10.2.2. Software Improvement of TTS

Several software synchronization methods that use the basic test&set and test primitives have been developed to reduce the bus traffic generated by processors spin-waiting for a lock variable (to enter a critical section). The amount of bus traffic generated is proportional to the number of processors actively contending for the lock variable, or the queue length of the critical section [28]. To reduce the contention a processor can delay itself for a certain amount of time after each failed attempt, before rejoining the queue of the critical section, as in the *collision avoidance* method [6]. The potential contention in a single critical section can be spread to multiple critical sections by creating a hierarchy (a tree) of lock variables, as in the *tournament locks* method [32]. But the most promising has been the *software queue* method [6, 32, 48]. The software queue method enqueues all processors that try to enter the same critical section, with the head of the queue being the processor that is currently in the critical section. When a processor finishes the critical section, it dequeues itself and notifies only the next processor in the queue, by modifying a lock variable that is only shared by the two processors. Since all other waiting processors spin wait on their own lock variables from private caches, no bus access will be generated from these processors. The locking procedure for the critical section (the prolog and epilog of a critical section operation) can actually be removed because exclusive access is automatically guaranteed by the software queue.

The software queue method does not eliminate the use of critical sections. In order to enqueue the spinning processors of the original critical section, a separate *sequencing* critical section is created to sequence, or order the processors. The sequencing critical section still needs to be implemented with some hardware/software synchronization mechanism. Since a small critical section has to be created the software queue method is effective only when the original critical section is large, and there is a significant amount of contention for it.

10.2.3. Direct Implementation of Critical Section: Lock Table

10.2.3.1. Motivation and Function Description

Cache coherence protocols can be broadly divided into two categories: *write broadcast* [47, 66], and *invalidation* protocols [29, 38, 50]. Write broadcast cache coherence protocols have been found to be superior to write invalidation protocols [69, 70]. In these papers analytical models are used and the probabilistic input dictates that all processors to have equal access to the shared data. Therefore shared data are more likely subjected to fine grain sharing, in which the write-broadcast should perform well. A similar result is also in [8], where the write broadcast protocols are found to do well when the contention to shared data is high.

Write broadcast does worse when the shared data are not subjected to fine grain sharing (or high contention). Therefore if we can distinguish between the shared data that are subjected to fine grain sharing from those that are not, one strategy is to use the right cache coherence protocol for each type of the shared data [14]. This has become the basic idea behind the competitive snooping [37], where a write broadcast protocol is switched to write-invalidate when the break even point in bus-related coherency traffic between the two protocols is reached. In this method the sharing pattern of a data is dynamically determined during the run time of a program. The sharing pattern of shared variables, however, can sometimes be determined during compile time based on the expected usage of the variables. One most prominent example is the *lock* variables, which are subjected to high contention. Thus accesses to the lock variables can benefit greatly from the write broadcast capability of the system. Allowing broadcast capability to lock variables is the major idea behind the *lock table* method.

In the lock table method when a processor wants to enter a critical section, it puts its *intention*, i.e., entering a critical section, along with an *identification of the critical section*, or the address of the lock variable that protect the critical section, on the bus. All other processors that want to enter the same critical section can decide to block their bus accesses after seeing this bus transaction. When a processor wants to leave a critical section, it gains access to the bus and puts the identification of the critical section on the bus. All processors waiting to enter the critical section will be wakened by this bus transaction, and try to get access to the bus. Only the first attempt will succeed, and the remaining processors simply block themselves again. So entering or leaving a critical section takes only one instruction (enter- or leave-critical-section) and one bus access (to broadcast the identification of the critical section).

This leaves one problem to fix: if a processor tries to enter a critical section that has been occupied earlier by another processor, this processor should know the status of the critical section to avoid an unnecessary bus access. This information is stored in a hardware structure called **lock table (LT)**. A lock table has multiple entries, each of which can be in either the *valid* or the *invalid* state. The lock table controller snoops on the bus only for the enter- and leave-critical-section events. When an enter-critical-section access appears on the bus an entry of the lock table is allocated. The identification of the critical section is stored in the entry and the entry is set to valid. When a leave-critical-section access appears on the bus, the valid entry with the same identification of the critical section is invalidated (freed). Every processor has a private copy of the lock table, and all copies contain the same information. When a processor wants to enter a critical section, it checks the local lock table first. If the critical section is not occupied, the processor will try to put its enter-critical-section access on the bus. If the critical section is occupied, or become occupied before the processor's access can be put on the bus, the processor will be blocked. The processor will not try to access the bus until the entry in its lock table is invalidated, i.e., the critical section is freed.

10.2.3.2. Implementation Issues

A lock table can be implemented like a snooping cache using a write broadcast protocol, so most of the cache design considerations apply. The first important issue is the *size* of, or the number of entries in, a lock table. A lock table is a system wide shared resource, meaning each critical section being occupied by a process in the system needs an entry in the lock table. Valid entries can belong to different parallel programs, even to those that have been temporarily swapped out of the multiprocessor. A suitable size may be found by projecting the maximum need for the table. But any size table can overflow, i.e., when all the entries become valid and some requests to enter new critical sections are pending.

A simple solution is to block the processes that try to use new entries. A more sophisticated way is to spill the entries into main memory. But a trap mechanism is required to transfer the entries, and to bring them back when they become active again. The operating system also has to manage buffer space in main memory for the spilled entries. Finally, as a last resort certain programs can always be aborted to release some entries. But caution is needed to avoid severely degrading of the system throughput. Table overflow may also result in deadlock, if a process can be in multiple critical sections at the same time. Aborting some programs may be the only way to break the deadlock. But care must be taken when the table overflow trap occurs either the operating system does not use the lock tables, or certain number of lock table entries have been reserved for the operating system so that free lock entries are always available to the trap operations.

A very different solution to the table overflow problem is to implement the lock tables as write broadcast caches. The most important advantage of using caches to implement lock tables is that the normal hardware cache replacement mechanism can take care of the table overflow problem, with much better performance than the trap mechanism. But this solution requires some changes to the semantics of the enter- and leave-critical-section operations. Now the status of a critical section is not known to a processor unless the cache line that contains the status of the lock variable for the critical section is in its cache. The cache miss ratio can always be reduced by using the traditional approaches for ordinary data caches, such as increasing the cache size or using multi-way set associative caches. Note that since the block size of the lock caches can be different from the one for the ordinary data caches, and the block size can be the smallest possible (it only has to store the status of the lock variable, essentially one bit information), bus traffic generated from lock cache misses is not as large as that from data cache misses.

Another important concern is the speed of *table lookup*. The tradeoff is similar to that of an ordinary cache. Using a large or fully associative table makes more efficient use of the entries, but table lookup may be slow. Using a small or direct mapped table makes faster lookup, but increases the chance of table overflow. Table overflow is a more serious problem than cache overflow because a processor can be blocked by table overflow, whereas a processor can proceed after a cache entry is replaced.

10.2.3.3. Generality and Scalability

The lock table method can be used to implement any critical section, which has very general use for processor synchronization. In fact all important synchronization needs can be by critical sections, though some can be more efficiently implemented using other methods. The most famous example is combining. A special version of the combining mechanism will be discussed in the next section.

The lock table can easily be adapted to any large scale, shared memory multiprocessor of a different architecture. All that is required is an interconnection network that has the broadcast capability. If the interconnect is dedicated to carrying only the enter-critical-section and leave-critical-section messages, the bandwidth requirement of the interconnect can be much lower than that of the interconnect that carries ordinary data traffic. In fact, even though implemented as a single bus that connects all processors, it has potential to support a much larger number of processors than an ordinary, single shared bus multiprocessor.

10.2.3.4. Comparisons to Other Synchronization Schemes

10.2.3.4.1. Semaphore Registers

The idea of using special hardware to assist the critical section operation is also used in the *semaphore register* method [55]. The differences between the lock tables and semaphore registers are in many ways similar to those between the ordinary data caches and data registers. For example, the lock tables are architecturally independent, hence the size of the lock tables can be increased at any time to improve the system performance without recompiling the programs. Other differences between the data caches and data registers such as the need for optimizing register allocation, and the registers being part of the process context, can also find the analogy between the lock tables and semaphore registers. Each of these differences may favor either of the two methods (data cache vs. data registers, and lock tables vs. semaphore registers), but in the case of lock tables and semaphore registers, overall the differences seem to make the lock table method a more favorable choice, as explained below.

One important issue with registers is register allocation. Allocation of the data registers can be done (optimized) at compile time to maximize the performance of a uniprocessor system. In a multiprocessor system if multiple parallel programs are allowed to be active at the same time, it is not easy to allocate the semaphore register (also at compile time) to optimize the overall system performance. One solution is to employ the idea of implementing large number of *virtual* or *logical* locks with a smaller number of hardware locks, as in Sequent Balance multiprocessors [67]. Since multiple virtual semaphore registers can be guarded by one hardware semaphore register, many more virtual semaphore registers are available and the allocation problem is alleviated. But the performance of the virtual semaphores is not so good as the hardware ones.

If the performance degradation of virtual semaphore registers is too much, large number of hardware semaphore registers may be needed because these semaphore registers can not be used efficiently. With the lock table method a table entry is used only when it is actually needed. A semaphore register, however, may be tied up by a parallel program during the whole course of program execution, whether the register is being actively used or not.

Since the program may spend only a small fraction of total time inside a critical section, many more semaphore registers are needed to achieve the same performance level as the lock table method when multiple parallel programs are allowed to run simultaneously in the system.

The use of semaphore registers also complicates the multiprogramming strategy in a multiprocessor. A parallel program, or any process of the program can be swapped out whether it holds an entry in the lock tables or not. With the semaphore register method the contents of the registers are part of the program state, and all processes that belong to the same program have to be swapped out to free the registers they use. When the program is scheduled to be swapped in, the operating system has to be sure that all the semaphore registers used by the program are in a free state.

Therefore the lock table method is more efficient and flexible than the semaphore register method. A simpler multiprocessor, in terms of both hardware and system software, can be built upon the lock table mechanism.

10.2.3.4.2. Queue_on_SyncBit (QOSB)

QOSB is a synchronization primitive that can be used to implement critical sections [31]. The primitive enqueues a processor to a lock variable (SyncBit), which is part of a cache line, to enforce a first-come first-serve discipline for the lock variable (and its associated critical section). The primitive was originally proposed for a large scale, cache-coherent shared memory multiprocessor Multicube [30], though it can be adapted to a simpler bus based multiprocessor.

The QOSB mechanism has inspired some variations. A similar but more complex hardware queueing mechanism is proposed to further implement non-exclusive critical sections by adding more cache line states [41]. The hardware queueing mechanism can also be simulated by software. The SyncBits in cache lines can be simulated by an array of lock variables [6], or a software queue data structure [48]. The major advantage of these software derivatives is that they can run on the existing multiprocessors which only support the simpler primitive of *test&set* and *compare&swap*, and immediately improve the performance of synchronization. However, their performance, such as the bus traffic generated, is not likely comparable to the hardware QOSB, especially for large multiprocessor configurations [73].

The comparisons between the lock table and QOSB methods are based on the following criteria: functionality, performance, implementation, and scalability. Both the lock table and QOSB methods directly support critical section operation. With the FCFS discipline the QOSB method further guarantees fair access to a critical section.[1] The lock table method, on the other hand, does not ensure fair access unless additional hardware support is available. The QOSB is also advisory in nature, i.e., it is used only to improve the performance of synchronization operations. The correct function of the operations is still guaranteed by the use of test&set. Since a QOSB access has a side effect of loading a cache line, when the QOSB access is non-blocking, it can be treated as a prefetching access.

Either the lock table or the QOSB method can generate more bus traffic than the other. Two bus operations are needed for a processor to go through a critical section when either method is used. With the lock table method one bus access is needed to enter and another to leave critical section; with the QOSB method one bus access is needed to enqueue and another to pass the lock/data to a waiting processor. But one side effect of acquiring a lock (SyncBit) in a QOSB operation is that the cache block containing the lock is also loaded into the cache. If the data in the cache block can be used by the processor, a likely situation when the shared data protected by the lock is put on the same cache block, the cost of the QOSB operation is essentially zero. This is an important optimization not because bus traffic is reduced, but because one non-local access is saved. In a large scale multiprocessor where the non-local access latency is likely large, reducing the number of non-local accesses may significantly improve the performance of the memory system.

However, if the data in the cache block can not be utilized by the processor, such as in database applications where one lock may have to protect a large amount of data, the QOSB access may create more bus traffic if the cache block size is large.

The performance of the QOSB method is also better than that of the lock table method if fair access to a critical section is important. For certain synchronization patterns, such as when multiple locks must be obtained to accomplish a computing task, fair access to the locks can limit the waiting time of a processor to obtain all the locks, letting more processors engage in productive work.

One problem with the QOSB method is that without a sophisticated compiler its potential performance advantage may not be easily exploited. For example, packing the right shared data with a SyncBit in a cache block is important to save bus traffic and reduce average memory access time. Taking advantage of the *prefetching* ability of the QOSB may also need to be carefully weighed against generating additional bus traffic by fetching cache lines

[1] The FCFS discipline is not strictly maintained because the hardware queue may be broken. This may occur when a cache lock that is part of the queue is replaced.

too early. Generating code for lock table mechanism, on the other hand, is straightforward because its functionality is more limited than the QOSB method. For the lock table method the traditional locking and unlocking procedures are simply replaced by the enter- and leave-critical section instructions.

It is hard to compare the hardware implementation costs between the two methods. While the amount of additional hardware needed to implement QOSB mechanism is easier to estimate (additional cache state bits, space for SyncBits, ...), the hardware cost of the lock table method is not clear because the optimal table size may depend on the size of the multiprocessors, and likely increase with the size of the multiprocessor.

The scalability of the QOSB method is probably better than that of the lock table method. Multiple QOSB requests can be outstanding in an interconnection network, taking full advantage of the parallelism offered by the interconnect. But the synchronization accesses (QOSBs) have to share the bandwidth with ordinary data traffic. The lock table, on the other hand, can rely on a separate synchronization interconnect, which carries only the synchronization signals (enter- and leave-critical-sections), for a large scale multiprocessor. But the bus bandwidth of the synchronization interconnect still limits the size of the multiprocessors.

Since the multiprocessor simulator running on Symmetry can not be adapted to simulate a synchronization mechanism that is drastically different from that in Symmetry, and the difference in performance of the lock table and QOSB methods can only be demonstrated when large scale multiprocessors are used with benchmarks that have high synchronization activities, no further performance study are conducted for the QOSB method in this thesis. Instead the performance of the lock table is compared with a software variation of the QOSB, the software queue method, which as described in an earlier section, can be implemented on Symmetry without additional hardware support.

10.2.4. Restricted Combining

For bus-based multiprocessors a restricted form of combining *fetch&*Φ operations was proposed [63]. The restricted form of combining allows an easy and efficient implementation, requiring only that all processors participating in the combining have an identical *fetch&*Φ operation. Although it can be used like test&set to implement critical sections, it actually obviates the need of a critical section when *the only operation in the critical section is a fetch&*Φ. The restricted form of combining *fetch&*Φ has a surprisingly large number of applications [27]. For example, in the parallel benchmarks all operations in scheduling critical sections can be converted into the restricted form of *fetch&decrement*[2]. Restricted combining also find its way into assisting the sequencing operation in the software queue method, improving the performance of large critical sections.

10.3. Performance Evaluation

The performance of the lock table and the restricted combining synchronization methods is evaluated using my multiprocessor simulator. During simulation, the simulator is capable of detecting the presence of a critical section in the parallel program code. The original code sequence of test&test&set, which is generated by the Symmetry compiler, is not executed. Instead the simulator generates a series of events to simulate the function of either the lock table or the restricted fetch&add (with combining) mechanism.

However, not every critical section can be converted in this way to utilize the new hardware synchronization mechanisms. Restricted combining is applicable only when the single operation in a critical section is, for example, a fetch&decrement. In the benchmarks only the scheduling critical section can be benefited from restricted combining For any other kind of critical sections the default test&test&set is used. The only such case occurs in the communication critical section of the benchmark SOR.

Restricted combining can also improve the barrier operation by combining the atomic decrement instructions. But the performance improvement may not be significant, because the atomic decrement instruction is so short that multiple processors are not likely to reach a barrier and execute the instructions at the same time to let combining occur. (Moreover, as shown in Chapter 9, the barrier operation itself is not much of a concern anyway.)

As mentioned earlier, the software queue method is useful only when a critical section is large. Since no critical section in the benchmarks is large (with simple dynamic self-scheduling), new versions of the benchmark FFT, MSORT and TREE with larger scheduling critical section are created. In the new version the Simple-Self Scheduling strategy, which schedules one loop iteration at a time, is changed to Guided-Self-Scheduling (GSS) [54], which uses heuristics to schedule multiple loop iterations at a time. The program code with the GSS scheduling critical section is then modified so that a software queue can be generated at run time. Note that the three benchmarks are chosen because in these benchmarks the amount of computation in a loop iteration is very small, and does not increase with the problem size. So they are more likely to benefit from the use of the GSS method [12, 54].

[2]Assuming the critical section implements the *simple-self-scheduling* policy, i.e., allocating one loop iteration at a time.

I will first discuss the performance improvement resulting from using lock tables or restricted combining instead of test&test&set. I'll then consider the performance of the software queue method, and how its performance is further improved by the use of the restricted combining mechanism.

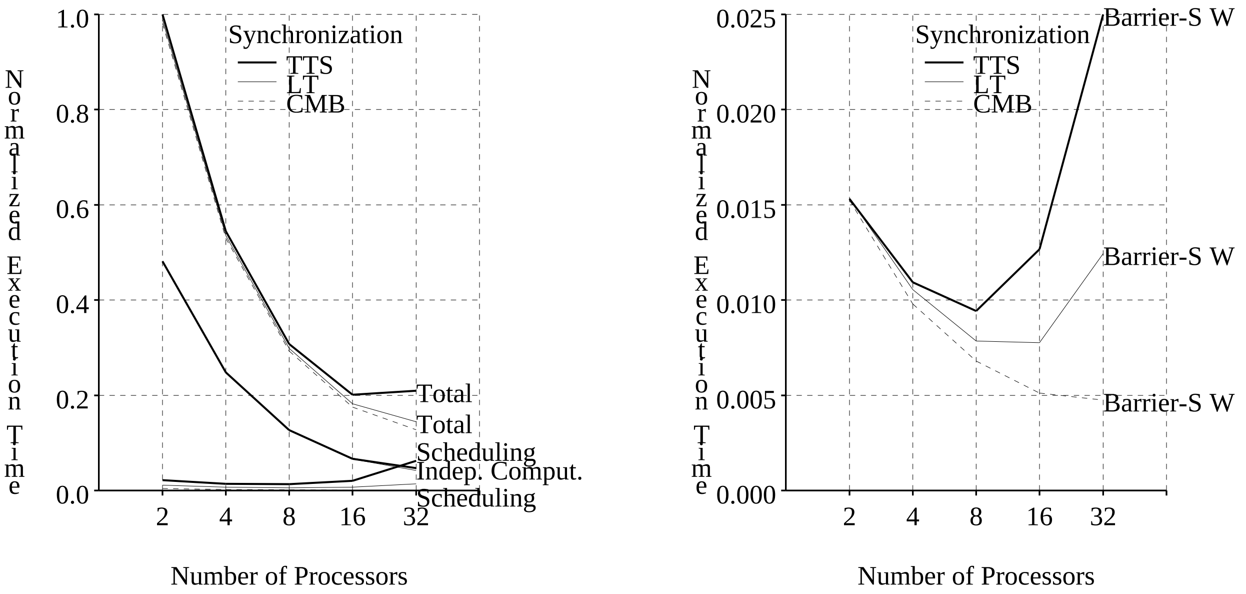
10.3.1. Simulation Results

10.3.1.1. Scheduling Phase

Since both the lock table and the restricted combining methods affect the implementation of critical sections, their effects on performance are most noticeable in the execution times of the critical sections. Figures 10.1(a), 10.2(a), 10.3(a), 10.4(a) and 10.5(a) show the normalized execution times of the scheduling phase, together with the times of the total and independent computing phases, for the three synchronization methods. In these five figures the execution time of the scheduling phase using the lock table mechanism decreases initially, and then increases with the number of processors. This speedup characteristic is similar to that when test&test&set is used. But the performance with the lock table is much better, especially when the number of processors is large. For example, when 32 processors are used the scheduling time is reduced by 77% for GAUSS. The reductions in scheduling times is 84%, 71%, 86%, 86% for the benchmarks FFT, SOR, MSORT, and TREE respectively, when the number of processors is 64. The performance improvement in the scheduling phase accounts for almost all the reduction in the total execution time, as depicted in these figures. For the above multiprocessor configurations, the reduction in the scheduling time results in 31%, 73%, 53%, 79% and 77% decrease in the total execution time of the benchmarks GAUSS, FFT, SOR, MSORT, and TREE respectively.

The lock table method improves scheduling efficiency mainly by forbidding bus access from processors waiting to be scheduled. But the serial nature of scheduling with the lock table method, however, can still make the scheduling rate the performance bottleneck when many processors are used. This shortcoming is completely avoided in restricted combining. As shown in the same five figures, even when 64 processors are used, the execution time of the scheduling phase is always nearly negligible. More interestingly, since each processor goes through the scheduling phase fewer number of times when more processors are used, the scheduling time of each processor actually *decreases* with the number of processors (though not clearly shown in these figures).

With the capability of scheduling all processors at the same time by the restricted combining method, the speedup of a parallel program now depends entirely on the bus bandwidth. But does this mean, with a fixed bus



(a) Total, Scheduling and Independent Computing Phases

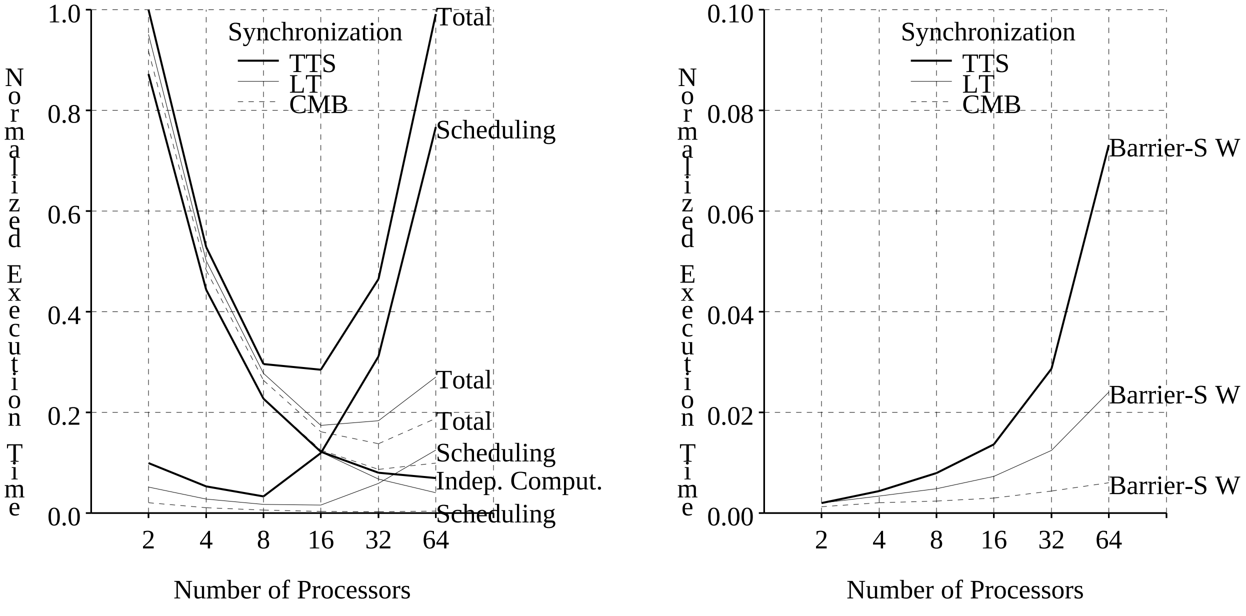
(b) Barrier-Spin Wait Phase

Figure 10.1: Normalized Execution Times of the Benchmark GAUSS

This figure, together with Figure 10.2, Figure 10.3, Figure 10.4 and Figure 10.5, shows the performance improvement made by the use of the *lock table* and the *restricted combining* synchronization methods. These two methods are used to implement the scheduling critical section, which uses the simple dynamic self-scheduling policy, in all benchmarks. The lock table method, which is applicable to any critical section, is also used to implement the communication critical section in the benchmark SOR. The results generated by using these two methods are compared with those that only use *test&test&set* in these figures.

The most direct performance improvement is, undoubtedly, in the execution times of the scheduling and communication phases. The improvement is more pronounced in the benchmarks FFT, MSORT and TREE, where the scheduling critical section is more burdened with scheduling smaller quanta of computational work. The total execution times are included in these figures to show that the performance improvement on scheduling is almost the only reason for the higher speedup of the entire program.

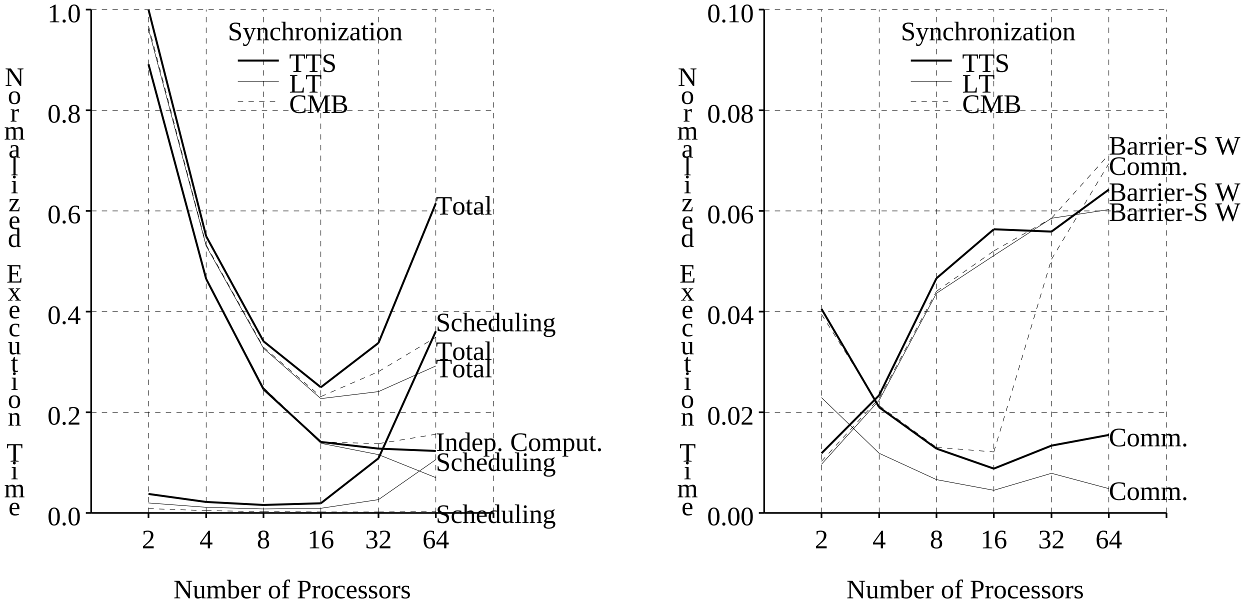
The better scheduling performance also indirectly affects the execution time of the independent computing and the barrier-spin wait phases. The execution time of the independent computing phase is always highest with the restricted combining method, and the lowest with the lock table method. The execution time of the barrier-spin wait phase, on the other hand, is always smaller with any of the two methods.



(a) Total, Scheduling and Independent Computing Phases

(b) Barrier-Spin Wait Phase

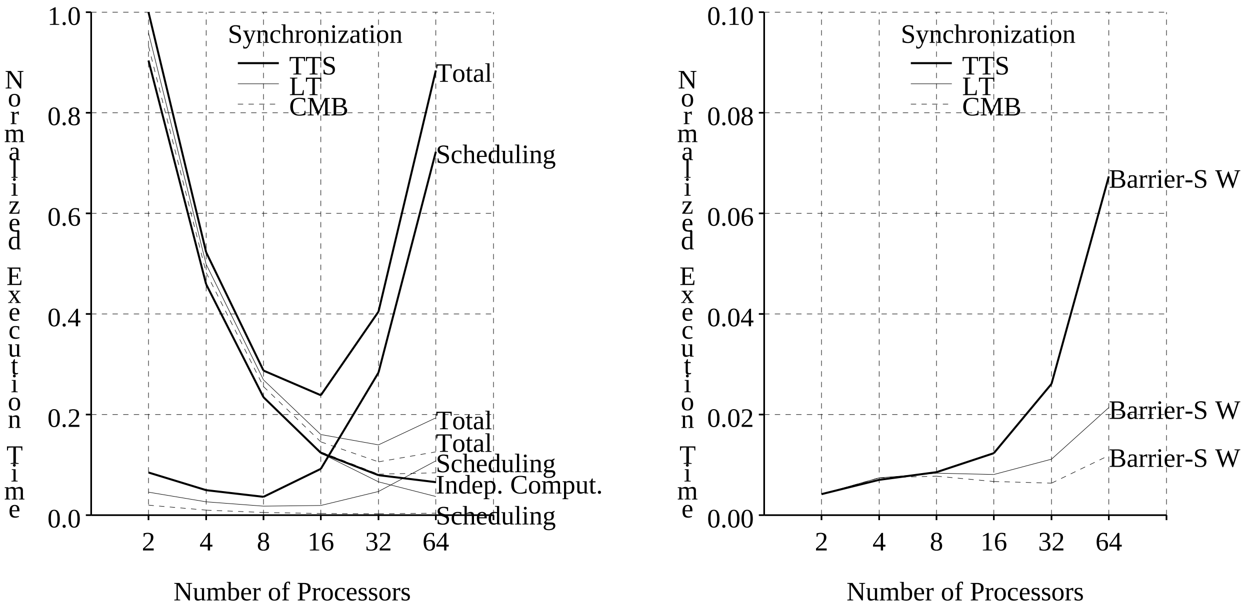
Figure 10.2: Normalized Execution Times of the Benchmark FFT



(a) Total, Scheduling and Independent Computing Phases

(b) Barrier-Spin Wait & Comm. Phases

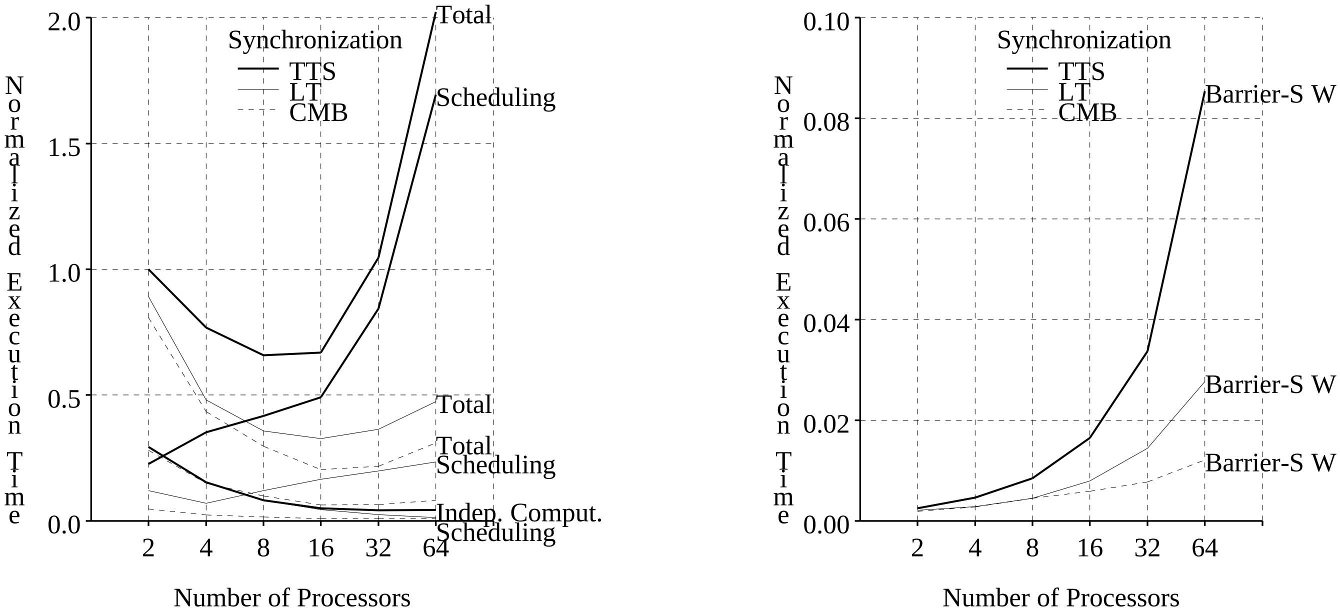
Figure 10.3: Normalized Execution Times of the Benchmark SOR



(a) Total, Scheduling and Independent Computing Phases

(b) Barrier-Spin Wait Phase

Figure 10.4: Normalized Execution Times of the Benchmark MSORT



(a) Total, Scheduling and Independent Computing Phases

(b) Barrier-Spin Wait Phase

Figure 10.5: Normalized Execution Times of the Benchmark TREE

bandwidth, that the best way to achieve the highest speedup of a parallel program is to use as many processors as possible to guarantee that the bus bandwidth is always fully utilized? The answer is no, as indicated by the speedup characteristics of the total execution time in the figures. For all except the benchmark GAUSS, with the restricted combining mechanism the total execution time is always higher with 64 processors than with only 32 processors. As suggested by Figures 10.2(a), 10.3(a), 10.4(a) and 10.5(a), the major cause for the decreased speedup lies at the independent computing phase. In these figures the execution times of the independent computing phase all increase when the number of processors is increased from 32 to 64 processors. In the next section the cause of this increase is explained.

10.3.1.2. Independent Computing Phase

Although the different synchronization mechanisms do not change the hardware resource requirement such as the bus demand of the independent computing phase, they still affect the execution time of the phase. As shown in Figures 10.1(a), 10.2(a), 10.3(a), 10.4(a) and 10.5(a), the execution time of the independent computing phase is always the highest with restricted combining, and the lowest with the lock table method. The different effects of synchronization mechanisms result from two factors: the *scheduling rate* and the *bus demand*.

When the scheduling rate is higher more processors can engage in independent computing at the same time. With restricted combining virtually *all* processors are in the independent computing phase at the same time. But as shown earlier, the total bus demand from all processors always increases with the number of processors (see Section 8.3.2). Since it is the bus bandwidth that determines the maximum speedup in the independent computing phase, and the bus has only a fixed bandwidth, the execution time of this phase can only become larger when the total bus demand increases.

The scheduling rate is limited when either the test&test&set or the lock table method is used to implement the scheduling critical section. But a major difference between these two methods is that, with the lock table method the waiting processors at the scheduling critical section *do not* generate any bus traffic, while with the test&test&set method these processors *do*. In other words, when the lock table method is used only those processors that are in the independent computing phase make bus access; whereas when the test&test&set method is used, every processor, whether it is in the scheduling or the independent computing phase, always make access to the bus. Since the execution time of the independent computing phase is determined by the bus utilization, and the bus utilization is lower with the lock table method, the execution time of the independent computing phase is also lower when the lock table method is used.

One interesting result of using the lock table method is that the execution time of independent computing phase can sometimes *decrease* after the scheduling critical section is saturated. This phenomenon is seen in Figure 10.3(a), and is explained as follow. After the critical section is saturated only a fixed number of processors can be in the independent computing phase at any time. The exact number is determined by the scheduling rate of the lock table method, and the rate is independent of the queue length of the critical section. Since it is always the same number of processors that are using the shared bus (doing independent computation), the execution time of one loop iteration will not change too much. But the share of computation, or the number of loop iterations for each processor in the independent computing phase decreases with the number of processors. Therefore as long as the bus utilization is not too high the execution time of the independent computing phase can decrease with the number of processors.

10.3.1.3. Communication Phase

The benchmark SOR is the only program that has a communication critical section. This critical section can not benefit from the restricted combining method. Therefore even though the combining mechanism is available, the critical section is still implemented by test&test&set. The execution time of the communication phase is shown in Figure 10.3(b). It is interesting to note that, executing the critical section takes longer with restricted combining, even though the critical section is always implemented with test&test&set. This is because with combining processors are scheduled in groups. Hence they are more likely to reach the communication critical section at the same time. Since the critical section is implemented with test&test&set, when the queue length of the critical section becomes large the execution time increases rapidly.

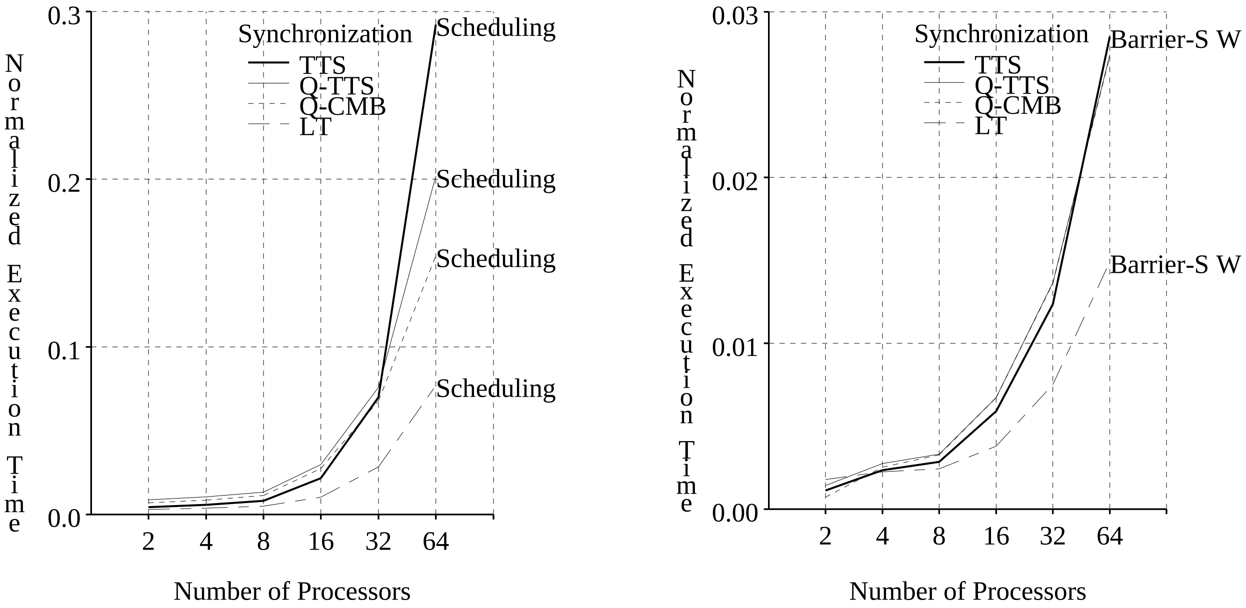
10.3.1.4. Barrier and Barrier-Spin Wait Phase

The use of different synchronization methods can also affects the barrier operation. In the barrier phase, where a barrier counter is decremented atomically, the restricted combinging method can improve the performance if combining occurs. But as discussed earlier, if an efficient atomic increment instruction is available combining is not likely to happen frequently. Since the execution time of the barrier phase is not significant even without combining, and the lock table method does not affect the operation of the atomic instruction, the effect of different synchronization mechanisms on the barrier phase is not discussed.

However, the use of different synchronization methods can significantly affects the execution time of the barrier-spin wait phase, as indicated in Figures 10.1(b), 10.2(b), 10.3(b), 10.4(b) and 10.5(b). In all figures the execution time is always the lowest when restricted combining is used, and highest when only test&test&set is available. Although the barrier operation can benefit from restricted combining, its effect is very small because the original atomic operation is so efficient that not much combining can happen. The major reason for the difference in the execution time is, with the more efficient scheduling method (i.e., higher scheduling rate), processors are more likely to start, thus finish independent computing and reach the barrier at the same time. The reduction in the execution time of this phase, however, does not affect the total execution time much, because the barrier-spin wait time has not been significant anyway.

10.3.1.5. Software Queuing

To evaluate the performance of a software queue, four combinations of software and hardware methods are selected for comparison in the following way. Scheduling implemented with the software queue method includes a sequencing critical section, for sequencing entries in the software queue. The critical section can then be supported with any of the three synchronization mechanisms (TTS, LT, and CMB). However, a critical section implemented with a software queue *always executes more instructions and makes more bus accesses* than the original scheduling critical section implemented with the lock table method. This is because with the LT method only two bus accesses are made (enter and leave-critical-section) to go through a critical section. With the software queue method at least one bus access is made at the sequencing critical section, independent of the hardware mechanism used to support the critical section, and another one made when one processor tries to wake up the next processor in the software queue. Therefore the case of using a software queue with the lock table method is not interesting. The performance of the remaining two cases, i.e., the software queue method with TTS for the sequencing critical section, and the software queue with CMB, is then compared with the cases where the scheduling critical section is supported by TTS (the base case) and LT respectively. The simulation results are shown in Figure 10.6, Figure 10.7 and Figure 10.8, for the benchmarks FFT, MSORT and TREE respectively. Only the execution times of the scheduling and the barrier-spin wait phases are shown in the figures. It is apparent from these figures that, if TTS is the only available synchronization mechanism, the software queue scheduling method always performs better, when many processors are used. This result is similar to what was reported in [6, 32]. The result here also shows that the performance can be further improved if the sequencing critical section is supported by the restricted combining mechanism. This



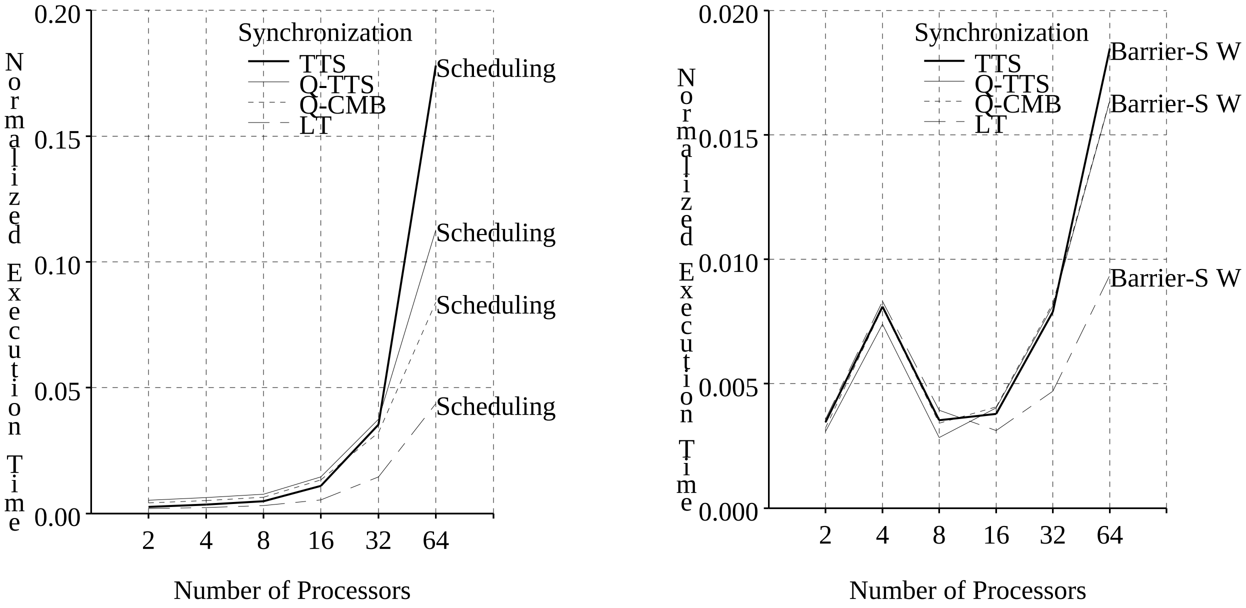
(a) Scheduling Phase

(b) Barrier-Spin Wait Phase

Figure 10.6: Normalized Execution Times of the Benchmark FFT

This figure together with Figure 10.7 and Figure 10.8 shows the performance improvement when the scheduling critical section is implemented with software queue. The scheduling strategy is Guided-Self Scheduling, which schedules multiple loop iterations at a time to each processor. The four cases considered in the figures are: 1, test&test&set (TTS), the base case; 2, software queue with TTS for the sequencing critical section (Q-TTS); 3, software queue with restricted combining for the sequencing critical section (Q-CMB); and 4, lock table (LT). In this figure as well as in the other two the normalizing factor for the benchmark is the execution time when only two processors are used, and critical sections implemented with TTS. As indicated by all three figures the rank of performance is always, from the best to the worse: LC, Q-CMB, Q-TTS, TTS.

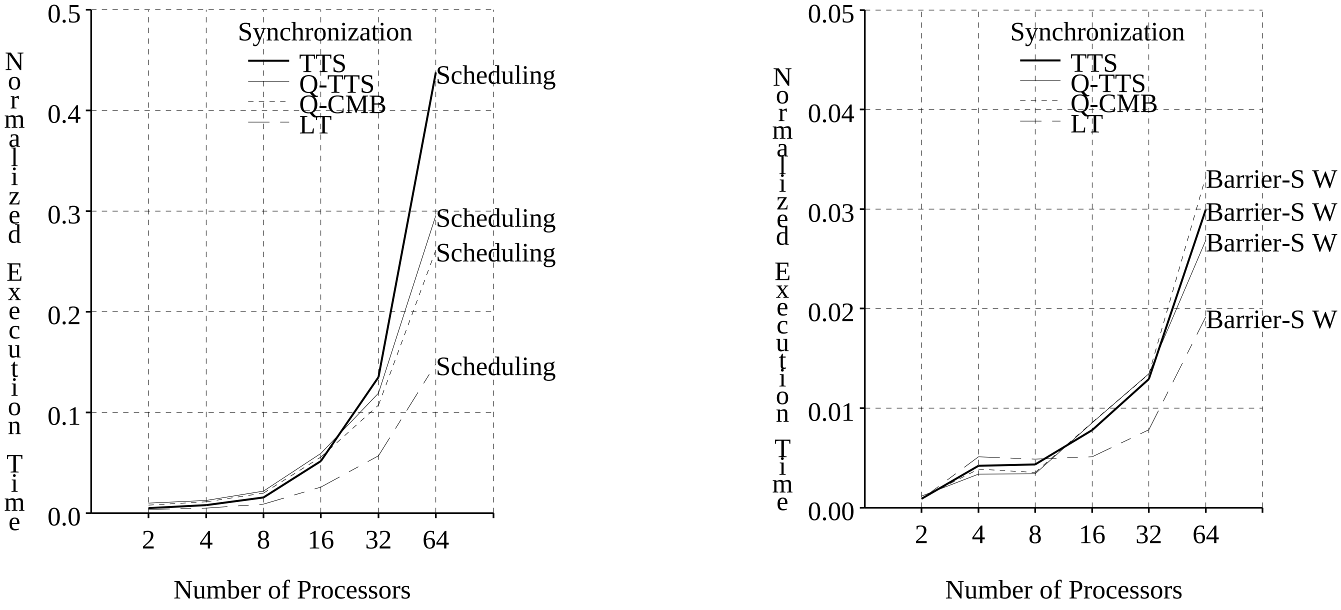
The performance of scheduling also affects the execution time of the barrier-spin wait phase, as discussed earlier. The LC method always reduces the barrier-spin wait time significantly when many processors are used.



(a) Scheduling Phase

(b) Barrier-Spin Wait Phase

Figure 10.7: Normalized Execution Times of the Benchmark MSORT



(a) Scheduling Phase

(b) Barrier-Spin Wait Phase

Figure 10.8: Normalized Execution Times of the Benchmark TREE

indicates that, even with a much smaller sequencing critical section than the original scheduling critical section, contention to the sequencing critical section can still be a problem when many (in my experiment, 64) processors are used.

The one that always performs the best, however, is the lock table method, no matter how many processors are used. The result is not surprising, since it only testifies the earlier theoretical result that the lock table method should always outperform the software queue method.

One thing to note is that, when restricted combining is available, there is little need to use guided-self-scheduling strategy. In fact simple-self-scheduling supported by restricted combining can outperform any other scheduling strategy supported by any combination of the software and hardware mechanisms. The catch is, however, that scheduling is not the only place where the performance bottleneck can occur. The generality of the lock table, or the software queue method is still needed to tackle with the type of critical sections that can not benefit from restricted combining.

10.3.1.6. Invalidation Pattern

In this section the invalidation pattern of using different synchronization methods are studied. Analyzing the invalidation pattern is important in evaluating different *directory schemes* for large scale, non-bus based, cache coherent multiprocessors [4, 72].

Previous study of invalidation patterns uses either analytical modeling [17, 21], or trace driven simulation [4, 72]. One drawback of using the trace driven simulation method is that, since the timing of the memory system to be evaluated is unlikely to be the same as the timing of the system where the trace is derived, the result of trace-driven simulation can be inaccurate. The problem is not so much due to the difference in *the timing* (or *interleaving*) *of the memory traces used*, as to the difference in *the type and the number of memory traces that should be used*. The most notable example that can cause this problem is the part of memory trace generated by *synchronization* operations.

Both the type and the number of memory accesses generated by a synchronization operation is highly variable, depending on the exact timing of the memory system components. If the memory accesses generated by synchronization operations have significant influence on the outcome of the simulation result, the result can be very different even with a small change to the memory system. As will be shown in my execution-driven simulation, it is not uncommon that more than 90% of total invalidations are generated by synchronization operations.

However, to some extent the result of trace driven simulation is still useful, since the input trace at least represents one realistic case. But the simulation result can be more appropriately interpreted if we can discern the part of the result that is highly variable from the part that is more stable. This information is available from my simulation, as shown in Table 10.1. The distribution of the numbers is represented by five entries in the table. The first three entries, representing 1, 2, or 3 actual invalidations of cache copies per invalidation request on the bus, show the effectiveness of a directory scheme that uses 1, 2 or 3 address pointers for each cache block. The remain-

Number of of Invalidations	Distribution - TTS & LT & CMB (in %)								
	GAUSS			FFT			SOR		
	TTS	LT	CMB	TTS	LT	CMB	TTS	LT	CMB
1	67	27	19	76	8.5	0.32	79	13	30
2	23	22	21	4.1	2.6	2.5	2.6	0.09	0.8
3	0.6	0	0	6.3	5.6	5.5	1.1	0.04	0.8
4-29	8.8	0	0	14	0	0	16	3.8	11
30-31	0.4	1.4	1.4	0.3	0.27	0.27	0.6	0.72	0.7
Total	**100**	**50**	**41**	**100**	**17**	**8.6**	**100**	**17**	**43**

Number of Invalidations	Distribution - TTS & LT & CMB (in %)					
	MSORT			TREE		
	TTS	LT	CMB	TTS	LT	CMB
1	79	9.0	0.19	83	9.4	1.9
2	3.3	0.86	0.80	2.5	0.16	0.26
3	1.7	1.0	1.1	0.82	0.13	0.20
4-29	16	2.2	2.2	13	1.1	1.4
30-31	0.16	0.17	0.17	0.40	0.13	0.15
Total	**100**	**13**	**4.5**	**100**	**11**	**3.9**

Table 10.1: Distribution of the Number of Actual Invalidations; Using Test&Test&Set, Lock Table or Restricted Combining

This table shows, when an invalidation request is put on the bus, how many cache copies are *actually* invalidated. All results are derived from the 32-processor configurations, and the synchronization primitive can be test&test&set (TTS), lock table (LT) or restricted combining (CMB). Results listed for the lock table method are generated by implementing *all* critical sections with the lock table mechanism, and the results for restricted combining (CMB) by using it only for the scheduling critical section. Therefore even though the combining mechanism is available, the communication critical section in SOR is still implemented with test&test&set.

The distribution of the number of invalidations is shown by listing the *normalized* number of actual invalidations in five categories: 1, 2, 3, 4-29, and 30-31 invalidations. The normalizing base for each benchmark is the total number of invalidation requests generated when the benchmark is run with test&test&set. Therefore the **Total** in this table for TTS is always 100% (all numbers are in percentage).

ing two entries (4 to 29, and 30 to 31) show the number of invalidations that results in broadcasts if at most three address pointers for each cache block are maintained. A separate entry of the 4 to 29 range is used to indicate the number of broadcasts that invalidate only a limited number of cache copies.

The most impressive result is that the total number of invalidations is reduced, and significantly for the benchmarks FFT, MSORT and TREE, by the use of either the lock table (LT) or the restricted combining (CMB) synchronization method (instead of test&test&set (TTS)). In these benchmarks the invalidation traffic is reduced by more than 80% with the LT method, and more than 90% with the CMB method. Since the purpose of a better synchronization method is to reduce the number of bus accesses, including those caused by invalidations during synchronization operations, the results in Table 10.1 imply that more than 90% of invalidations are generated by synchronization operations. This means that the result of invalidation pattern analysis can be overwhelmingly dominated by the invalidations generated from synchronization operations. As discussed earlier, the invalidation pattern, mainly the number and the type of invalidations, can vary widely with small changes in the timing of multiprocessors.

The reduction in invalidations is less spectacular for the benchmark GAUSS with either method. This is because with only 32 processors, synchronization or scheduling in GAUSS has not been a performance bottleneck. The improvement in the benchmark SOR is also not so impressive with the CMB method, because the communication critical section in SOR can not utilize the CMB mechanism, and is still implemented with the default TTS. But the result of using LT in benchmark SOR is as good as for benchmarks FFT, MSORT and TREE.

Since very little bus traffic is generated from synchronization operations when either the LT or the CMB method is used, the results derived from the two methods are quite insensitive to the timings of synchronization events. Therefore the results can readily be generalized to any kind of cache coherent, bus or non-bus based multiprocessors. The enormous reduction in invalidation traffic as indicated in Table 10.1 attests to the much better scalability of multiprocessors that use the two better synchronization methods, than the ones that only use the TTS method.

10.4. Summary and Conclusion

In this chapter the performance of several software/hardware synchronization alternatives to the simple test&test&set (TTS) method is evaluated. The emphasis is put on two hardware mechanisms: the lock table (LT) method, which was introduced in this thesis, and the restricted combining (CMB) method. With the two better synchronization mechanisms the execution time of scheduling can be reduced significantly when compared to the simple TTS. The LT and the CMB methods also affect the execution time of the independent computing and the barrier-spin wait phases, but on a much smaller scale.

The performance of a software queue, with its sequencing critical section supported by either the TTS or the CMB mechanism, was also evaluated. The result shows that software queues with CMB can perform significantly better than with TTS when many processors (e.g., 64) are used. However, theoretically the LT can always outperform the software queue method, regardless of how the sequencing critical section is implemented.

Finally the invalidation pattern of using the different TTS, LT and CMB mechanisms are studied. Both LT and CMB can reduce, and sometimes significantly, the number of invalidations. This result also implies that the study of invalidation patterns based on memory traces generated with the TTS method may be unreliable, since the majority of invalidations can be generated by synchronization operations. Both the number and the type of invalidations (differentiated, for example, by the number of actual cache copies invalidated per bus invalidation) generated by synchronization operations can be very sensitive to small changes in the timings of the memory system.

The LT method appears to be a very promising synchronization method for large scale, cache coherent, bus or non-based multiprocessors. This is due to the generality of its function (applies to any critical section, which is a very general synchronization primitive), and its scalability (a separate interconnect for the LT function carries substantially less traffic than the interconnect for data).

Chapter 11
Summary and Conclusions

This dissertation studied the design of single bus, shared memory multiprocessors. The goal of the studies was to find optimum points in the design space for different memory system components that include private caches, shared bus and main memory. To this end new methodologies were developed to evaluate accurately the system performance, and applied to evaluate some key parameters in the design space.

The performance of a multiprocessor is strongly affected by its workload. The exact information available from the workload also determines the type of tools suitable for evaluating the performance. From a preliminary study two different types of workloads were identified. The first type of workload consists of independent, multi-user tasks, and each task is executed by only one processor. The performance of a multiprocessor operating with this workload is measured by the overall throughput of the multiprocessor; the operating environment of the multiprocessor is *throughput-oriented* under this type of workload. The second type of workload consists of parallel programs, each of which has multiple cooperating tasks running on multiple processors to solve a single problem; the performance of a multiprocessor operating with this type of workload is measured by the speedup of each parallel program, and the multiprocessor is operating in a *speedup-oriented* environment. These two environments are not mutually exclusive, i.e., both types of workload can exist at the same time in a multiprocessor. But the nature of the two types of workload is sufficiently different that totally different approaches were developed to evaluate the multiprocessor performance for each environment.

11.1. Throughput Oriented Environment

For the throughput-oriented environment the *Customized Mean Value Analysis* (CMVA) method was used to evaluate the performance of multiprocessors. Two separate CMVA models were developed for the two kinds of shared bus architectures, *circuit switched* and *split-transaction, pipelined*, or *STP* buses, to evaluate the multiprocessor performance. The two models were then validated using actual trace-driven simulation for about 5,376 multiprocessor configurations, constructed by varying the number of processors, processor speed, cache size, cache block size, cache set-associativity, bus width, bus switching method and main memory latency.

For the three workloads (two are non-homogeneous and one is homogeneous) in this operating environment at least 90% of all cases the percentage difference in individual processor throughputs (64,512 comparisons from the

5,376 multiprocessor configurations) obtained from the models and simulation is within 3%. The maximum difference in all cases is always less than 15%. The comparison results on the partial bus utilization of individual processors are also similar. Estimation of the cache miss latency is less accurate than the estimation of processor throughput or bus utilization, but is still quite good. Since the execution time of the CMVA models are about 4 to 5 orders of magnitude less than trace-driven simulation, the models were used to explore the design space extensively for shared bus, throughput-oriented multiprocessors.

The results obtained from the CMVA models have shown that the goal of designing the caches for a multiprocessor is not to minimize only the miss ratio or the bus traffic. For example, the best block size that maximizes multiprocessor throughput is neither the one that results in the lowest cache miss ratio, nor the one that minimize the bus traffic. Furthermore, the design of one memory component can not be considered independent of other components. With the circuit switched bus, the maximum system throughput is strongly affected by the main memory latency, while with the STP bus the maximum throughput is quite insensitive to the change in main memory latency. The performance of the STP bus can also be significantly improved, though with diminishing returns, by increasing the bus width.

Since the maximum system throughput is determined by the per-processor bus bandwidth demand, using a larger cache size or set associative caches can improve the system performance significantly, even when the same strategy has only a marginal effect on uniprocessor performance. For example, using the workload from the ATUM trace a 2-way set associative cache is found to perform worse than a direct mapped cache for a uniprocessor system, when the cache size is as large as 256K bytes and the speed of the direct mapped cache is 10% faster. But the same 2-way set associative cache is found to be more desirable than the direct mapped cache in multiprocessors.

It should be noted that the evaluation results presented in this thesis are valid only for the multiprocessor configurations and the workloads considered in the study. While certain conclusions may still be true when some of these conditions change, accurate performance results of a particular multiprocessor system should still be obtained by constructing a set of CMVA models that precisely describe the structure of the multiprocessor system, and by using the parameters of the expected workload for the multiprocessor to drive these models.

11.2. Speedup Oriented Environment

For the speedup-oriented environment a multiprocessor simulator was developed to conduct execution-driven simulation so that accurate and detailed performance information could be obtained. Since the execution of a parallel program consists of different computational phases with distinctive speedup characteristics, these phases were delineated, and performance statistics were collected separately for each computational phase. The performance statistics, which include the execution time and the cache miss ratios of various types of memory accesses, were then analyzed to determine the speedup characteristics of a particular computational phase.

Although accurate results can be obtained from execution driven simulation, a major drawback is that the simulation is very slow. Therefore only a small region of the design space was explored. In particular only the number of processors and the synchronization mechanisms were changed in the multiprocessor configuration. Therefore the research concentrated on two goals: 1) to study the way the memory access patterns change with the number of processors, and 2) to evaluate the performance of some synchronization mechanisms suited for bus-based shared memory multiprocessors.

It should be noted that since a very small part of the design space of the multiprocessors is explored in this thesis conclusions of this thesis are valid only for the multiprocessor configurations that have been considered.

The results obtained from simulating the execution of five parallel benchmarks have shown that, with loop level parallelization, the speedup of a parallel program is largely determined by the execution times of the independent computing phase and, if dynamic scheduling is used, the scheduling phase. The maximum speedup of the independent computing phase is limited by the bus bandwidth. But after reaching the maximum speedup by using enough processors, the speedup can decrease if more processors are used. This is because the total bus demand, or the total number of bus accesses made by all processors during this phase, can increase with the number of processors. The major causes of the increase are fine grain data sharing, reference spreading, and the use of dynamic scheduling. Therefore even for this computational phase, perfect speedup, i.e., cutting the execution time in half by doubling the number of processors, is difficult to achieve because the average memory access time seen by a processor is also increased due to the increase of the total bus demand.

Bus demand can usually be reduced by using a static scheduling strategy, especially if fine grain data sharing can be totally avoided with this strategy. Bus demand is also affected by the problem size. If the total bus demand grows slower than the total amount of computation as the problem size increases, the maximum speedup can increase with the problem size.

The scheduling overhead can limit the speedup in the independent computing phase even before the bandwidth of the shared bus. Whether scheduling can become a performance bottleneck depends on the relative size of the scheduling overhead to the scheduling unit. With loop level parallelization, and the scheduling unit being a loop iteration, only one or two dozen processors can actually be scheduled to do independent computation simultaneously, if test&test&set is used to implement the scheduling critical section. The size of a loop iteration may or may not grow with the problem size. Therefore if dynamic scheduling is used to exploit loop-level parallelization, the synchronization strategy needs to be streamlined.

The execution times spent in other computational phases such as the serial, the barrier, and the barrier-spin wait phases are usually relatively small in our benchmarks. For the barrier operation if an atomic decrement instruction such as the one in the Sequent Symmetry is available, the execution time of the barrier operation itself, i.e., excluding the spin-wait time due to different processor arrival times to the barrier, is negligible. The proportion of these execution times become even smaller when the problem size increases.

To improve the performance of synchronization, the *lock table* method was proposed in this thesis. The motivation of the lock table method is that a write broadcast cache coherence protocol is particularly suited for the access to lock variables. The performance of the lock table method, along with the restricted fetch&add (with combining) and the software queue method, were evaluated and compared against the basic test&test&set. The results have shown that both the lock table and the restricted combining methods can significantly reduce the bus traffic during the critical section operation, mainly scheduling, that otherwise would be generated by the test&test&set implementation. Moreover, the restricted combining method also allows parallel access to the scheduling critical section. The execution time of the scheduling phase is thus significantly reduced. For the 32 processor configuration of the benchmark GAUSS and the 64 processor configuration of the benchmarks FFT, SOR, MSORT and TREE, with the lock table mechanism replacing the test&test&set the scheduling times are reduced by at least 70%, resulting in 31% to 79% decreases in the total execution times. For the same multiprocessor configurations that used the restricted fetch&add with combining mechanism instead of the test&test&set the scheduling times become virtually negligible, and the total execution times are reduced by 39% to 86%.

The restricted combining method essentially eliminates the performance degradation due to scheduling by allowing parallel access to the scheduling critical section, but its functionality is more limited than that of the lock table method. The lock table, though not performing as well as the restricted combining method in reducing the scheduling overhead, has a larger application domain than that of the restricted combining method.

11.3. Future Work

The CMVA models developed in this thesis can be further expanded to include certain hardware optimization for a shared bus multiprocessor. For example, the performance of different bus arbitration policies, or the effect of loading requested data first in a cache block and bypass it to processors, can be evaluated. The models can be modified, or completely new models can be developed for the hierarchical (multi-level) cache memory system, which may assume a more important role as the disparity between the speed of processors and the memory system widened.

The multiprocessor simulator developed for execution-driven simulation has provided many important insights to the execution of parallel processing. But the simulator can provide much more information than has been discussed in this thesis. To extend the work, more and diverse parallel benchmarks should be developed, especially the ones that have *doacross* loops. Since the only multiprocessor configuration parameters that have been varied are the number of processors and the hardware synchronization mechanisms, most of the design space of the multiprocessors remains to be explored. For example, an important issue is the choice of the cache block size, and the related choice of bus width. The study of the cache block is compounded by the false sharing problem. Since the effect of false sharing can be substantial, software and hardware optimization methods must be developed to reduce the problem before tackling the cache block size problem.

The lock table synchronization method proposed in this thesis deserves further study, especially its implementation cost and effectiveness in a bus based, or a non-bus based, large scale shared memory multiprocessor. Since using the lock table does not preclude the use of other type of high performance hardware synchronization mechanism such as restricted combining, better performance might be achieved for a multiprocessor that employs these two methods. The performance of the lock table method should also be evaluated in this context.

Finally, a multiprocessor is not likely to be used solely for one particular environment. A multiprocessor designed for the throughput-oriented environment needs an operating system to manage independent processes. The operating system kernel in all aspects is a parallel program. A multiprocessor that is speedup-oriented may also like to have multiple parallel programs running at the same time to increase multiprocessor throughput. It would be the final challenge to design multiprocessor that is optimum in both operating environments.

Acknowledgements

I have long awaited the chance to express my deepest appreciation to those who helped me through the years of my graduate study, culminating in the writing of this dissertation. I owe the most to Dr. Guri Sohi, my academic advisor. Not only did I receive a tremendous amount of technical help from him in my research (including English writing), but also I learned from him a mature attitude toward independent study, which is absolutely essential for a doctorate degree. Learning to have a truly independent attitude in research was especially difficult for me, given that my early years of education were strongly influenced by the Chinese education tradition that emphasizes subordination, even in academic thinking. I may not have learned all the right things that he has tried to teach me, but I am grateful that his efforts have already transformed me profoundly.

I would like to pay my deepest respects to Dr. Mary K. Vernon. She helped me to learn analytical modeling, which is a major part of my dissertation. Her insistence on accuracy in every detail of the methodology and precision in the result presentation, will be valuable in my future research efforts.

I appreciate the technical advice and information constantly provided by Dr. Mark Hill, and the parallel application provided by Dr. Renato De Leone. Special thanks to Michael Cheng, who proofread the final version of this dissertation and corrected many English mistakes. I am also grateful to the people working in the system lab; they offered timely help to smooth out the problems in using different computing facilities.

Bob Holloway and all the secretaries in the Computer Science Department deserve my hearty thanks. Bob helped me greatly during my long tenure as a teaching assistant, and those wonderful secretaries helped me avoid entanglement with administration.

I understand that it will be a futile effort to try to mention everyone that has make my life so memorable during these years as a graduate student in Madison. But once in a while some of these names will come back to my mind, and my appreciation to them will constantly be refreshed.

Abstract

This dissertation studies the design of single bus, shared memory multiprocessors. The purpose of the studies is to find optimum points in the design space for different memory system components that include private caches, shared bus and main memory.

Two different methodologies are used based on the operating environment of a multiprocessor. For a multiprocessor operating in the *throughput-oriented* environment, Customized Mean Value Analysis (CMVA) models are developed to evaluate the performance of the multiprocessor. The accuracy of the models are validated by comparing their results to those generated by actual trace-driven simulation over several thousand multiprocessor configurations. The comparison results show that the CMVA models can be as accurate as trace driven simulation in predicting the multiprocessor throughput and bus utilization. The validated models are then used to evaluate design choices that include cache size, cache block size, cache set-associativity, bus switching method, bus width, and main memory latency.

For a multiprocessor operating in the *speedup-oriented* environment a multiprocessor simulator is built to conduct an execution driven simulation for several parallel benchmarks. Performance statistics are collected separately for different computational phases during the execution of a parallel program, and are analyzed to characterize the behavior of the multiprocessor in each computational phase and the overhead of parallel processing. The results show that, with loop level parallelization, performance bottlenecks can occur in the *scheduling* phase (if dynamic scheduling is used to schedule loop iterations to processors) as well as the *independent computing* phase. The cache miss ratio of shared data in the independent computing phase is also found to increase with the number of processors, meaning that even for this phase the speedup will decrease instead of leveling off after the speedup has reached its maximum.

The barrier operation in a bus based multiprocessor is found not to be a serious concern. With an efficient atomic decrement instruction available, the total time for all processors to decrement the barrier counter can be very a small portion of overall execution. Both the barrier and serial times can become less significant, i.e., become a smaller portion of total execution time of a parallel program, when the problem size is increased.

The impact of the synchronization method on the execution time of the scheduling phase is also considered in some detail. The performance of several alternatives to the base test&test&set implementation, such as a software queue, a lock table (proposed in this thesis), and a restricted fetch&add operation with combining, are evaluated.

The simulation result shows that both the lock table and restricted combining methods can improve the scheduling rate significantly.

Table of Contents

List of Tables

Chapter 11: Summary and Conclusions

List of Figures

Bibliography

[1] ‘‘Introduction to the iAPX-432 Architecture,’’ *Intel Manual No. 171821-001*, 1981.

[2] A. Agarwal, R. L. Sites, and M. Horowitz, ‘‘ATUM: A New Technique for Capturing Address Traces Using Microcode,’’ in *Proc. 13th Annual Symposium on Computer Architecture*, Tokyo, Japan, pp. 119-127, June 1986.

[3] A. Agarwal, J. Hennessy, and M. Horowitz, ‘‘Cache Performance of Operating Systems and Multiprogramming Workloads,’’ *ACM Transactions on Computer Systems*, vol. 6, pp. 393-431, November 1988.

[4] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, ‘‘An Evaluation of Directory Schemes for Cache Coherence,’’ in *Proc. 15th Annual Symposium on Computer Architecture*, Honolulu, HI, pp. 280-289, June 1988.

[5] A. Agarwal, M. Horowitz, and J. Hennessy, ‘‘An Analytical Cache Model,’’ *ACM Transactions on Computer Systems*, vol. 7, pp. 184-215, May 1989.

[6] Thomas E. Anderson, ‘‘The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors,’’ *IEEE Trans. on Parallel and Distributed System*, vol. 1, January 1990.

[7] Marco Annaratone and Roland Ruhl, ‘‘Performance Measurements of a Commercial Multiprocessor Running Parallel Code,’’ in *The 16th International Symposium on Computer Architecture*, Jerusalem, Israel, pp. 307-314, May 1989.

[8] J. Archibald and J. -L. Baer, ‘‘An Evaluation of Cache Coherence Solutions in Shared-Bus Multiprocessors,’’ *ACM Transactions on Computer Systems*, vol. 4, pp. 273-298, November 1986.

[9] J. Archibald and J. -L. Baer, ‘‘High Performance Cache Coherence Protocols For Shared-Bus Multiprocessors,’’ Technical Report 86-06-02, Department of Computer Sciences, University of Washington, Seattle, WA 98195, June 1986.

[10] H. B. Bakoglu and T. Whiteside, ‘‘RISC System/6000 Hardware Overview,’’ in *IBM RISC System/6000 Technology*, Austin, TX, pp. 8-15, 1990.

[11] B. Beck, B. Kasten, and S. Thakker, ‘‘VLSI Assist for a Multiprocessor,’’ *Proc. ASPLOS II*, pp. 10-20, October 1987.

[12] Carl Josef Beckmann, ‘‘Reducing Synchronization and Scheduling Overhead in Parallel Loops,’’ CSRD Report Number 922, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL 61801, October 1989.

[13] C. G. Bell, ‘‘Multis: a New Class of Multiprocessor Computers,’’ *Science*, vol. 228, pp. 462-467, April 1985.

[14] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, ‘‘Adaptive Software Cache Management for Distributed Shared Memory Architectures,’’ in *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, Washington, pp. 125-134, May 1990.

[15] Anita Borg, R. E. Kessler, and David W. Wall, ‘‘Generation and Analysis of Very Long Address Traces,’’ *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 270-279, 1990.

[16] P. Borrill and J. Theus, ‘‘An Advanced Communication Protocol for the Proposed IEEE 896 Futurebus,’’ *IEEE Micro*, pp. 42-56, August 1984.

[17] L. M. Censier and P. Feautier, ‘‘A New Solution to Coherence Problems in Multicache Systems,’’ *IEEE Transactions on Computers*, vol. C-27, pp. 1112-1118, December 1978.

[18] H. Cheong and A. V. Veidenbaum, ‘‘A Cache Coherence Scheme with Fast Selective Invalidation,’’ in *Proc. 15th Annual Symposium on Computer Architecture*, Honolulu, HI, pp. 299-307, June 1988.

[19] Conte and De Boor, *Elementary Numerical Analysis - An Algorithmic Aprroach*. New York: McGraw-Hill, 1980.

[20] R. Cytron, ‘‘Doacross: Beyond Vectorization for Multiprocessors (Extended Abstract),’’ *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 836-844, August, 1986.

[21] M. Dubois and F. A. Briggs, ''Effects of Cache Coherence in Multiprocessors,'' *IEEE Trans. on Computers*, vol. C-31, pp. 1083-1089, November 1982.

[22] S. J. Eggers and R. H. Katz, ''A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evalaution,'' in *Proc. 15th Annual Symposium on Computer Architecture*, Honolulu, HI, pp. 373-382, June 1988.

[23] S. J. Eggers and R. H. Katz, ''Evaluating the Performance of Four Snooping Cache Coherence Proctcols,'' *Proc. of 16th Int'l Symp. Computer Architecture*, pp. 2-15, 1989.

[24] S. J. Eggers and R. H. Katz, ''The Effect of Sharing on the Cache and Bus Performance of Parallel Programs,'' in *Proc. ASPLOS-III*, Boston, MA, pp. 257-270, April 1989.

[25] G. N. Fielland, ''Symmetry: A Second Generation Practical Parallel,'' *Digest of Papers, COMPCON Spring 1988*, pp. 114-115, February 1988.

[26] S. J. Frank, ''Tightly Coupled Multiprocessor System Speeds Memory-Access TImes,'' *Electronics*, vol. 57, 1, pp. 164-169, January 1984.

[27] Eric Freudenthal and Allan Gottlieb, ''Process Coordination with Fetch-and-Increment,'' in *ASPLOS-IV Proceedings*, Santa Clara, CA, pp. 260-268, April, 1991.

[28] Andy Glew and Wen-Mei Hwu, ''Snoopy Cache Test-and-Test-and-Set Without Excessive Bus Contention,'' *Computer Architecture News*, vol. 18, pp. 25-32, June 1990.

[29] J. R. Goodman, ''Using Cache Memory to Reduce Processor-Memory Traffic,'' *Proc. 10th Annual Symposium on Computer Architecture*, pp. 124-131, June 1983.

[30] J. R. Goodman and P. J. Woest, ''The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor,'' in *Proc. 15th Annual Symposium on Computer Architecture*, Honolulu, HI, pp. 422-431, June 1988.

[31] J. R. Goodman, M. K. Vernon, and P. J. Woest, ''A Set of Efficient Synchronization Primitives for a Large-Scale Shared-Memory Multiprocessor,'' in *Proc. ASPLOS-III*, Boston, MA, April 1989.

[32] Gary Graunke and Shreekant Thakkar, ''Synchronization Algorithms for Shared-Memory Multiprocessors,'' *Computer*, vol. 23, pp. 60-69, June 1990.

[33] Rajiv Gupta, ''The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors,'' in *Proc. ASPLOS IV*, Santa Clara, CA, April 1991.

[34] M. D. Hill, ''Aspects of Cache Memory and Instruction Buffer Performance,'' Technical Report UCB/CSD 87/381, University of California at Berkeley, Berkeley, CA, November 1987.

[35] M. D. Hill, ''A Case for Direct-Mapped Caches,'' *IEEE Computer*, vol. 21, pp. 25-40, December 1988.

[36] R. Jog, G. S. Sohi, and M. K. Vernon, ''The TREEBus Architecture and Its Analysis,'' Computer Sciences Technical Report #747, University of Wisconsin-Madison, Madison, WI 53706, February 1988.

[37] A. Karlin and et al, ''Competitive Snoopy Caching,'' *Proc. of 27th Ann. Symp. Foundations of Computer Science*, pp. 244-254, 1986.

[38] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, ''Implementing a Cache Consistency Protocol,'' *Proc. 12th Annual Symposium on Computer Architecture*, pp. 276-283, June 1985.

[39] T. Lang, M. Valero, and I. Alegre, ''Bandwidth of Crossbar and Multiple-Bus Connections for Multiprocessors,'' *IEEE Transactions on Computers*, vol. C-31, pp. 1227-1234, December 1982.

[40] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance, Computer System Analysis Using Queueing Network Models*. Englewood Cliffs, New Jersey: Prentice Hall, 1984.

[41] J. Lee and U. Ramachandran, ''Synchronization with Multiprocessor Caches,'' in *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, Washington, pp. 27-37, May 1990.

[42] R. De Leone and O. L. Mangasarian, ''Asynchronous Parallel Successive Overrelaxation For The Symmetric Linear Complementarity Problem,'' Computer Sciences Technical Report #755, University of Wisconsin-Madison, Madison, WI 53706, Feb. 1988.

[43] S. Leutenegger and M. K. Vernon, ''A Mean-Value Performance Analysis of a New Multiprocessor Architecture,'' *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* May 1988.

[44] D. Lilja, D. Marcovitz, and P.-C. Yew, ''Memory Reference Behavior and Cache Performance in a Shared Memory Multiprocessor,'' CSRD Report Number 836, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL 61801-2932, December 1988.

[45] M. A. Marsan and M. Gerla, ''Markov Models for Multiple-Bus Multiprocessor Systems,'' *IEEE Transactions on Computers,* vol. C-31, pp. 239-248, December 1982.

[46] M. A. Marsan, G. Balbo, G. Conte, and F. Gregoretti, ''Modeling Bus Contention and Memory Interference in a Multiprocessor System,'' *IEEE Transactions on Computers*, vol. C-32, pp. 60-72, January 1983.

[47] E. McCreight, ''The DRAGON Computer System: An Early Overview,'' in *NATO Advanced Study Institute on Microarchitecture of VLSI Computer*, Urbino, Italy, July 1984.

[48] J. M. Mellor-Crummey and M. L. Scott, ''Algorithmms for Scalable Synchronization on Shared-Memory Multiprocessors,'' *ACM Transactions on Computers Systems*, vol. 9, pp. 21-65, February 1991.

[49] T. N. Mudge, J. P. Hayes, G. D. Buzzard, and D. C. Windsor, ''Analysis of Multiple Bus Interconnection Networks,'' *Proc. 1984 International Conference on Parallel Processing*, pp. 228-232, August 1984.

[50] M. S. Papamarcos and J. H. Patel, ''A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories,'' *Proc. 11th Annual Symposium on Computer Architecture*, pp. 348-354, June 1984.

[51] J. H. Patel, ''Analysis of Multiprocessors With Private Cache Memories,'' *IEEE Transactions on Computers*, vol. C-31, pp. 296-304, April 1982.

[52] G. F. Pfister and V. A. Norton, '''Hot-Spot' Contention and Combining in Multistage Interconnection Networks,'' *IEEE Transactions on Computers*, vol. C-34, pp. 943-948, October 1985.

[53] C. C. Polychronopoulos, ''On Program Restructuring, Scheduling and Communication for Parallel Processor Systems,'' in *Ph. D. dissertation, Center for Supercomputing Research and Development*, University of Illinois at Urbana-Champaign, August 1986.

[54] C. D. Polychronopoulos and D. J. Kuck, ''Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers,'' *IEEE Transaction on Computer*, December 1987.

[55] Richard D. Pribnow, ''System For Multiprocessor Communication Using Local and Common Semaphore and Information Register,'' *U.S. Patent No. 4,754,398*, June 1985.

[56] S. Przybylski, M. Horowitz, and J. Hennessy, ''Performance Tradeoffs in Cache Design,'' *Proc. 15th Annual Symposium on Computer Architecture*, pp. 290-298, June 1988.

[57] L. Rudolph and Z. Segall, ''Dynamic Decentralized Cache Schemes for MIMD Parallel Processors,'' *Proc. 11th Annual Symposium on Computer Architecture*, pp. 340-347, June 1984.

[58] Michael L. Scott and John M. Mellor-Crummey, ''Synchronization without Contention,'' in *Proc. ASPLOS IV*, Santa Clara, CA, April 1991.

[59] Y. Shiloach and U. Vishkin, ''An O(log n) parallel connectivity algorithm,'' *J. Algorithms*, pp. 57-63, 1982.

[60] R. L. Sites and A. Agarwal, ''Multiproccessor Cache Analysis Using ATUM,'' in *Proc. 15th Annual Symposium on Computer Architecture*, Honolulu, HI, pp. 186-195, June 1988.

[61] A. J. Smith, ''Cache Memories,'' *ACM Computing Surveys*, vol. 14, pp. 473-530, Sept. 1982.

[62] A. J. Smith, ''Line (Block) Size Choice for CPU Cache Memories,'' *IEEE Transactions on Computers*, vol. C-36, pp. 1063-1075, September 1987.

[63] Gurindar S. Sohi, James E. Smith, and James R. Goodman, ''Restricted Fetch&Φ Operations for Paralle Processing,'' in *Proc. 3rd International Conference on Supercomputing*, Crete, Greece, June 1989.

[64] P. Tang and P.-C. Yew, ''Processor Self-Scheduling for Multiple-Nested Parallel Loops,'' *Proceededings of the 1986 International Conference on Parallel Processing*, pp. 528-535, August 1986.

[65] P. Tang, P.-C. Yew, and C.-Q. Zhu, ''Impact of Self-Scheduling Order on Performance of Multiprocessor Systems,'' in *ACM International Conferece of Supercomputing*, Malo, France, pp. 593-603, July 1988.

[66] C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr., ''Firefly: A Multiprocessor Workstation,'' *IEEE Transactions on COmputers*, vol. C-37, pp. 909-920, August 1988.

[67] S. Thakkar, P. Gifford, and G. Fielland, ''Balance: A Shard Memory Multiprocessor System,'' *Proc. 2nd Int. Conf. on Supercomputing*, pp. 93-101, 1987.

[68] Kishor Shridharbhai Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Application*. Prentics-Hall Inc., 1982.

[69] M. K. Vernon and M. Holliday, ''Performance Analysis of Multiprocessor Cache Consistency Protocols Using Generalized Timed Petri Nets,'' *Proc. SIGMETRICS International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pp. 9-17, May 1986.

[70] M. K. Vernon, E. D. Lazowska, and J. Zahorjan, ''An Accurate and Efficient Performance Analysis Technique for Multiprocessor Snooping Cache-Consistency Protocols,'' in *Proc. 15th Annual Symposium on Computer Architecture*, Honolulu, HI, pp. 308-315, June 1988.

[71] M. K. Vernon, R. Jog, and G. S. Sohi, ''Performance Analysis of Hierarchical Cache-Consistent Multiprocessors,'' *Performance Evaluation*, vol. 9, pp. 287-302, 1989.

[72] Wolf-Dietrich Weber and Anoop Gupta, ''Analysis of Cache Invalidation Patterns in Multiprocessors,'' in *Proceedins ASPLOS-III*, Boston, MA, April, 1989.

[73] P. J. Woest and J. Goodman, ''An Analysis of Synchronization Mechanisms in Shared-Memory Multiprocessors,'' in *Proceedings of the International Symposium of Shared Memory Multiprocessing*, Tokyo, Japan, pp. 152-165, April 1991.

[74] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie, ''Distributing Hot-Spot Addressing in Large Scale Multiprocessors,'' *IEEE Transactions on Computers*, vol. C-36, pp. 388-395, April 1987.

[75] John Zahorjan, Edward D. Lazowska, and Derek L. Eager, ''The Effet of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems,'' July 1989.